Matching and Compression of Strings with Automata and Word Packing

Skjoldjensen, Frederik Rye

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Matching and Compression of Strings with Automata and Word Packing

Frederik Rye Skjoldjensen

PHD-2017-446, February 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, Building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

PHD-2017-446
ISSN: 0909-3192

Preface

This doctoral dissertation was prepared at the Department of Mathematics and Computer Science at the Technical University of Denmark as partial fulfillment of the requirement for acquiring a doctoral degree. The work presented in this dissertation was done from March 2014 to February 2017 under the supervision of Associate Professor Philip Bille and Associate Professor Inge Li Gørtz. The dissertation is comprised of two peer-reviewed publications, a submitted paper and a unpublished paper. The project was funded by the Danish Research Council (DFF – 1323-00178).

Acknowledgement First and foremost, I would like to thank my supervisors Philip Bille and Inge Li Gørtz for giving me the opportunity of experiencing academia from the perspective of a researcher. You have pushed me forward academically and helped me digest some of the idiosyncrasies of academia. I would also like to thank my great colleagues and office mates Hjalte, Søren, Patrick, Anders, Mikko and Nicola for fun and exciting times at the office. I had the great experience of visiting Professor Giuseppe F. Italiano at Università di Roma "Tor Vergata" for four months and I am very grateful that he took the time for discussions and social activities with me and his students. Lastly, I would like to thank my family, especially Agnete.

Frederik Rye Skjoldjensen
Lyngby, February 2017

Abstract

Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation Given a static reference string R and a source string S, a relative compression of S with respect to R is an encoding of S as a sequence of references to substrings of R. Relative compression schemes are a classic model of compression and have recently proved very successful for compressing highly-repetitive massive data sets such as genomes and web-data. We initiate the study of relative compression in a dynamic setting where the compressed source string S is subject to edit operations. The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. We present new data structures that achieve optimal time for updates and queries while using space linear in the size of the optimal relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates. To achieve these results, we revisit the dynamic partial sums problem and the substring concatenation problem. We present new optimal or near optimal bounds for these problems. Plugging in our new results we also immediately obtain new bounds for the string indexing for patterns with wildcards problem and the dynamic text and static pattern matching problem.

Subsequence Automata with Default Transitions Let S be a string of length n with characters from an alphabet of size σ. The *subsequence automaton* of S (often called the *directed acyclic subsequence graph*) is the minimal deterministic finite automaton accepting all subsequences of S. A straightforward construction shows that the size (number of states and transitions) of the subsequence automaton is $O(n\sigma)$ and that this bound is asymptotically optimal.

In this paper, we consider subsequence automata with *default transitions*, that is, special transitions to be taken only if none of the regular transitions match the current character, and which do not consume the current character. We show that with default transitions, much smaller subsequence automata are possible, and provide a full trade-off between the size of the automaton and the *delay*, i.e., the maximum number of consecutive default transitions followed before consuming a character.

Specifically, given any integer parameter k, $1 < k \leq \sigma$, we present a subsequence automaton with default transitions of size $O(nk \log_k \sigma)$ and delay $O(\log_k \sigma)$. Hence, with $k = 2$ we obtain an automaton of size $O(n \log \sigma)$ and delay $O(\log \sigma)$. At the other extreme, with $k = \sigma$, we obtain an automaton of size $O(n\sigma)$ and delay $O(1)$, thus matching the bound for the standard subsequence automaton construction. Finally, we generalize the result to multiple strings. The key component of our result is a novel hierarchical automata construction of independent interest.

Deterministic Indexing for Packed Strings Given a string S of length n, the classic string indexing problem is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. In the *deterministic* variant the goal is to solve the string indexing problem without any randomization (at preprocessing time or query time). In the *packed* variant the strings are stored with several character in a single word, giving us the opportunity to read multiple characters simultaneously. Our main result is a new string index in the deterministic *and* packed setting. Given a packed string S of length n over an alphabet σ, we show how to preprocess S in $O(n)$ (deterministic) time and space $O(n)$ such that given a packed pattern string of length m we can support queries in (deterministic) time $O\left(m/\alpha + \log m + \log\log\sigma\right)$, where $\alpha = w/\log\sigma$ is the number of characters packed in a word of size $w = \Theta(\log n)$. Our query time is always at least as good as the previous best known bounds and whenever several characters are packed in a word, i.e., $\log\sigma \ll w$, the query times are faster.

Dynamic Partial Sums in Constant Time and Succinct Space with the Ultra Wide Word-RAM Model The dynamic partial sums problem is to dynamically maintain an array of n integers while supporting efficient access, update and partial sums queries. This classic problem, and its variations, are very well studied in many different computational models [Fre82, FS89, Fen94, HSS11, HR03, HRS96, RRR01, PD04]. We solve the partial sums problem in the *ultra wide word-RAM model*, recently introduced by Farzan et al. [FLONS15], where we, in constant time, are allowed to manipulate words of size w^2 and access w memory locations. Farzan et al. [FLONS15] additionally gave a solution to the dynamic partial sums problem by simulating the RAMBO model to obtain a result by Brodnik et al. [BKMN06]. In this paper we present an improved solution to the dynamic partial sums problem in the ultra wide word-RAM model that supports all operations in either constant or $O(\log\log n)$ time, depending on whether we allow multiplication, and succinct space. We pose as an open problem whether it is possible in the ultra wide word-RAM model to additionally support the classic select operation in constant time.

Danish Abstract

Dynamisk relativ kompression, dynamiske delsummer og delstrengskonkatenering Givet en statisk referencestreng R og en kildestreng S, er en relativ kompression af S med hensyn til R en indkodning af S som en sekvens af referencer til delstrenge i R. Relativ kompression er en klassisk model for kompression der for nyligt har vist sig at være meget egnet til komprimering af ekstremt store mængder meget repetiv data som f.eks. genomer og webdata. Vi påbegynder studiet af relativ kompression i en dynamisk kontekst hvor vi tillader at kildestrengen kan ændres løbende. Målet er at vedligeholde den komprimerede repræsentation kompakt mens understøttelse for ændringer og tilfældigt opslag i den ukomprimerede kildestreng er bibeholdt. Vi præsenterer en ny datastruktur der, for næsten alle parametre, opnår optimal køretid for opdateringer og forespørgsler mens pladsforbruget er linært i størrelsen af den optimale relative kompression. Vi præsenterer derudover løsninger hvor vi begrænser og udvider de typer opdateringer vi understøtter. For at opnå disse resultater undersøger vi problemerne med at repræsentere dynamiske delsummer og svare på delstrengskonkatenerings forespørgsler. Vi præsenterer nye optimale og næsten optimale grænser for disse problemer. Hvis vi benytter disse resultater i streng indeksering for mønstre med jokertegn problemet og dynamisk tekst og statisk mønster genkendelses problemet opnår vi direkte bedre grænser for disse problemer.

Delsekvensautomater med standardtransitioner Lad S være en streng med længde n og tegn taget fra et alfabet med størrelse σ. Delsekvensautomaten bygget over S (ofte kaldet *directed acyclic subsequence graph* i litteraturen) er den minimale deterministiske automat der accepterer alle delsekvenser af S. En simpel konstruktion af automaten viser at størrelsen (antallet af tilstande og transitioner) af automaten er $O(n\sigma)$ og at denne grænse er asymptotisk optimal. I denne artikel undersøger vi delsekvensautomater med *standardtransitioner*, som er specielle transitioner der kun følges hvis ingen normale transitioner er identiske med det aktuelle tegn, og som ikke konsumere det aktuelle tegn. Vi viser at standard transitioner muliggør meget mindre delsekvensautomater, og giver en fuld afvejning mellem størrelse af automaten og *forsinkelsen* af automaten, som er det maksimale antal af på hinanden følgende standardtransitioner som vi kan følge før vi konsumere et tegn. Mere specifikt, givet en heltals parameter k, $1 < k \leq \sigma$, giver vi en delsekvensautomat med standardtransitioner med størrelse $O(nk \log_k \sigma)$ og forsinkelse $O(\log_k \sigma)$. På denne måde opnår vi med $k = 2$ en automat med størrelse $O(n \log \sigma)$ og forsinkelse $O(\log \sigma)$. I det andet yderpunkt, med $k = \sigma$, opnår vi en automat med størrelse $O(n\sigma)$ og forsinkelse $O(1)$, som er identisk med den oprindelige delsekvensautomat uden standard transitioner. Derudover generaliserer vi resultatet til automater over flere strenge. Nøglekomponenten i vores resultat er en original hierarkisk automat konstruktion der i sig selv er interessant.

Deterministisk indeks for pakkede strenge Det klassiske strengindekseringsproblem er, givet en streng S af længde n, at forhåndsprocessere S til en datastruktur der efterfølgende understøtter effektive forespørgsler efter mønstre. I den *deterministiske* kontekst er målet at løse strengindekseringsproblemet uden brug af randomisering (både ved forhåndsprocessering og forespørgsler). I den *pakkede* kontekst er strengene gemt med flere tegn i hvert maskinord så flere tegn kan læses simultant. Vores primære resultat er et nyt strengindeks i den deterministiske *og* pakkede kontekst. Givet en pakket streng S af længde n med tegn fra et alfabet af størrelse σ, viser vi hvordan S kan forhåndsprocesseres i $O(n)$ deterministisk tid og $O(n)$ plads så vi kan svare på forespørgsler på en pakket streng af længde m i deterministisk tid $O\left(m/\alpha + \log m + \log\log\sigma\right)$, hvor $\alpha = w/\log\sigma$ er antallet af tegn vi kan pakke i et maskinord af størrelse $w = \Theta(\log n)$. Vores forespørgselstid er altid mindst lige så god som den bedste tidligere grænse og når adskillige tegn kan pakkes i et maskinord, dvs. når $\log\sigma \ll w$, er vores forespørgsler hurtigere.

Dynamiske delsummer i konstant tid og succinct plads med Ultra Wide Word-RAM Modellen Det dynamiske delsumsproblem er at vedligeholde en dynamisk tabel med n heltal og samtidig understøtte effektive opslag, opdateringer og delsums forespørgsler. Dette klassiske problem, og dets variationer, er velstuderet i mange forskellige beregningsmodeller [Fre82, FS89, Fen94, HSS11, HR03, HRS96, RRR01, PD04]. Vi løser delsumsproblemet i *ultra wide word-RAM modellen*, fornyligt introduceret af Farzan et al. [FLONS15], hvor vi, i konstant tid, kan manipulere ord af størrelse w^2 og tilgå w hukommelseslokationer. Farzan et al. [FLONS15] gav desuden en løsning til det dynamiske delsumsproblem ved at simulere RAMBO modellen for at opnå et resultat af Brodnik et al. [BKMN06]. I denne artikel præsenterer vi en forbedret løsning til det dynamiske delsumsproblem i ultra wide word-RAM modellen hvor alle operationer understøttes i enten konstant tid eller $O(\log\log n)$ tid, afhængigt af om vi tillader multiplikation, og succinct plads. Som åbent problem stiller vi derudover spørgsmålet om hvorvidt det er muligt at understøtte den klassiske select operation i konstant tid.

Contents

Chapter 1

Introduction

This introduction will summarize the papers that I have coauthored during my studies and elaborate on some of the topics. The full papers are given in each of the following chapters. The dissertation consists of this introduction and the following (revised) papers.

Chapter 2 Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation. Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj and Søren Vind. In *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, Leibniz International Proceedings in Informatics (LIPIcs), Volume 64, 2016, pages 18:1–18:13.

Chapter 3 Subsequence Automata with Default Transitions. Philip Bille, Inge Li Gørtz and Frederik Rye Skjoldjensen. In *42nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2016)*, Lecture Notes in Computer Science, vol 9587, 2016, pages 208-216.

Chapter 4 Deterministic Indexing for Packed Strings. Philip Bille, Inge Li Gørtz and Frederik Rye Skjoldjensen. Submitted to *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*.

Chapter 5 Dynamic Partial Sum in Constant Time and Succinct Space with the Ultra Wide Word Model. Philip Bille, Inge Li Gørtz and Frederik Rye Skjoldjensen. Unpublished.

1.1 Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation

In Chapter 2 we study *dynamic relative compression*. Given strings S and R, we are interested in representing S as a sequence of references to substrings in R. We call S the *source string* and R the *reference string*. We define a *relative compression of S with respect to R* as a sequence $C = (i_1, j_1), \ldots, (i_{|C|}, j_{|C|})$ such that $S = R[i_1, j_1] \cdots R[i_{|C|}, j_{|C|}]$ and call C a *substring cover* or just a *cover* for S. Each element of C is a pair of indices representing a substring in R and when the substrings are concatenated together, in the order their indices appear in C, we obtain S. We say that C is an *optimal cover* of S if no other cover of S exists with fever references to R. The

dynamic relative compression problem is to maintain a relative compression of S under the following operations. Let i be a position in S and α be a character.

access(i): return the character $S[i]$,

replace(i, α): change $S[i]$ to character α,

insert(i, α): insert character α before position i in S,

delete(i): delete the character at position i in S.

In Chapter 2 we also consider two variations of the problem. In the first, we restrict the operations to access and replace. In the second, we maintain a *set* of strings $\mathcal{S}$ and, besides supporting the operations listed above on individual strings in $\mathcal{S}$, consider the operations concat and split between two strings in $\mathcal{S}$.

We assume a standard unit-cost RAM with w-bit words with standard arithmetic and logical operations on words and obtain the following result:

Theorem 1. *Let R and S be a reference and source string of lengths r and N, respectively, and let n be the length of the optimal substring cover of S by R. Then, we can solve the dynamic relative compression problem supporting* access, replace, insert, *and* delete

(i) in $O(n + r)$ space and $O\left(\frac{\log n}{\log \log n} + \log \log r\right)$ time per operation, or

(ii) in $O(n + r \log^{\epsilon} r)$ space and $O\left(\frac{\log n}{\log \log n}\right)$ time per operation, for any constant $\epsilon > 0$.

The result above is obtained under the assumption that $w \geq \log(n + r)$.

A critical part in obtaining the space bounds of Theorem 1 is that we are able to maintain a cover of S that contains no more than twice the number of references to R than an optimal cover of S. We do this by maintaining a cover C where we guarantee that the string obtained from concatenating the substrings of R represented by any two consecutive pairs of indices in C will never give a string that exists as a substring of R. We call a cover with this property for a *maximal cover* and in Lemma 1 we prove that a maximal cover is never more than twice the size of any other cover. Initially, we construct a maximal cover of S with respect to R by constructing the suffix tree of R and then greedily matching the maximal prefix of the remaining part of S: Assume that we have constructed the maximal cover C for the prefix $S[0..i]$. We then extend C by matching the maximal prefix of $S[i + 1..N]$ in the suffix tree of R. The next two sections describe the data structures we use for solving the dynamic relative compression problem. After every operation that changes S we use a *substring concatenation* data structure to maintain the maximality of the cover. When we answer queries on S we need to access the relevant part of R. We employ a *dynamic partial sums* data structure on the length of the substrings represented by C to access R fast. The substring concatenation data structure improves the best known bounds while the partial sums data structure introduce additional operations while maintaining the best known bounds.

1.1.1 Dynamic Partial Sums

Each operation on S implies an access to C for locating the correct substring of R. We call a pair of indices of C for a block and the length of a block is the length of the substring that it represents.

The block that contains index i of S is the first block of C for which the sum of the previous block lengths, including its own length, is larger than or equal to i. We store the block lengths of C in a partial sums data structure such that we can locate the block containing a given index fast. The partial sums problem is to maintain an integer array Z of length s under the following operations:

$\mathsf{sum}(i)$**:** return $\sum_{j=1}^{i} Z[j]$,

$\mathsf{update}(i, \Delta)$**:** set $Z[i] = Z[i] + \Delta$,

$\mathsf{search}(t)$**:** return $1 \leq i \leq s$ such that $\mathsf{sum}(i-1) < t \leq \mathsf{sum}(i)$. To ensure well-defined answers, we require that $Z[i] \geq 0$ for all i.

$\mathsf{insert}(i, \Delta)$**:** insert a new entry in Z with value Δ before $Z[i]$,

$\mathsf{delete}(i)$**:** delete the entry $Z[i]$ of value at most Δ.

$\mathsf{merge}(i)$**:** replace entry $Z[i]$ and $Z[i+1]$ with a new entry with value $Z[i] + Z[i+1]$.

$\mathsf{divide}(i, t)$**:** , where $0 \leq t \leq Z[i]$. Replace entry $Z[i]$ by two new consecutive entries with value t and $Z[i] - t$, respectively.

We obtain the following result:

Theorem 2. *Given an array of length s storing w-bit integers and parameter δ, such that $\Delta < 2^{\delta}$, we can solve the dynamic partial sums problem supporting* sum*,* update*,* search*,* insert*,* delete*,* merge*, and* divide *in linear space and $O(\log s / \log(w/\delta))$ time per operation.*

We store the cover C as doubly linked list and keep an auxiliary link from each block length stored in the partial sums data structure to the corresponding block in C. We can now answer an $\mathsf{access}(i)$ query on S by first finding the block number that contains index i by performing a $\mathsf{search}(i)$. We follow the auxiliary reference and obtain the block $b_l = (i_l, j_l)$ from C. The local index in $R[i_l, j_l]$ of the ith character in S is then $\ell = i - \mathsf{sum}(l-1)$ such that the answer to the query is character $R[i_l + \ell - 1]$. For the other queries, we locate $b_l = (i_l, j_l)$ and ℓ as for access, but since these queries change S we need to update the partial sums data structure and C. The details of how to perform the remaining operations are given in Section 2.2.2.

Our Solution The following is a brief description of our solution for solving the partial sums problem. Pătraşcu and Demaine [PD04] present a solution to the partial sums problem that we extend by introducing support for the operations insert, delete, merge and divide on Z. Our construction follows their ideas but we simplify their construction by taking advantage of a result by Pătraşcu and Thorup [PT14] for maintaining small sets of integers. The overall idea is to construct a solution for small sequences of $B = \Theta(\min(w/\log w, w/\delta))$ elements where all partial sums operations take constant time. This result is summarized in Theorem 6. We then use this solution for representing nodes in a B-tree that store Z in the leafs. The B-tree has height $O(\log_B s) = O(\max(\frac{\log s}{\log(w/\log w)}, \frac{\log s}{\log(w/\delta)})) = O(\log s / \log(w/\delta))$. A result by Willard [Wil00] enables us to maintain this tree under all partial sums operations in $O(\log s / \log(w/\delta))$ time per operations which gives us Theorem 2.

In Section 2.3 we give a more elaborate explanation of how we obtain Theorem 2, but neglect the details of how we maintain the auxiliary links between the block lengths stored in the partial

sums data structure and the indices of the blocks stored in the cover C. Our data structure for representing sequences of size B can be augmented with pointers to auxiliary data in the following way: We maintain an array x of size $O(B)$ where each index store $O(\log B)$ bits. We keep this array aligned with the array $\mathcal{C}$. When we obtain an index i from a search operation we can then access $O(\log B)$ bits of auxiliary data in x. The array x requires $O(B \log B)$ bits and can therefore be stored in a single word, just as with $\mathcal{C}$. To obtain the connection to the cover C we additionally maintain the dynamic array X that contains an index for each of the elements stored in the partial sums data structure. In X we store pointers to C. Given the base address of X, we only need $O(\log B)$ additional bits for locating an element in X. Hence, in each index of x we store an index into X that contains a pointer to an element of C.

1.1.2 Substring Concatenation

Every time we perform a replace, insert, or delete operation we make a change to S and therefore also need to update the cover. When we update the cover we must maintain the maximality property by checking that the concatenation of the changed block and either of the neighbor blocks is not occurring as a substring in R. We use a *substring concatenation data structure* for performing this check. The substring concatenation data structure answers the following query on R: Given two pairs of indices (i, i') (j, j'), does a third pair (k, k') exists such that $R[k, k'] = R[i, i']R[j, j']$. The query returns any pair (k, k') if it exists.

Before presenting our data structure we will first introduce previous solutions by Amir et al. [ALLS07] and Gawrychowski et al. [GLN14]. We need the following notation for suffix trees: Let T_S be the suffix tree of string S and let $I_S(v)$ denote the indices of the suffixes of S that are represented by the leafs in the subtree of node v in T_S. We additionally define $I'_S(v) = \{|S| - i + 2 \mid i \in I_S(v)\}$.

The scheme by Amir et al. Amir et al. [ALLS07] give a data structure for answering substring concatenation queries that is based on constructing the suffix trees of R and the reversal of R, R_{rev}, denoted as T_R and $\mathsf{T}_{R_{rev}}$, respectively. They observed that if we can find an index l of R from where we can match $R[i, i']$ backwards and $R[j, j']$ forwards then $l - (i' - i)$ is an answer to the substring concatenation query. More exact, we need to find an index l of R such that $R[l-1]R[l-2] \dots R[l-(i'-i)-1]$ match $R[i, i']_{rev}$ and $R[l]R[l+1] \dots R[l+(j'-j)]$ match $R[j, j']$. They do this by finding the intersection between all indices where we can match $R[i,' i]$ backwards and all indices where we can match $R[j, j']$ forwards. These two sets of indices are exactly $I'_{R_{rev}}(v_b)$ and $I_R(v_f)$, where v_b is the node found by matching $R[i, i']_{rev}$ in $\mathsf{T}_{R_{rev}}$, and v_f is the node found by matching $R[j, j']$ in T_R. We use $I'_{R_{rev}}(v_b)$ to transform the indices of forward matches of $R[i, i']_{rev}$ in R_{rev} to indices of backward matches of $R[i, i']$ in R. This transformation can be performed globally on the indices stored in the leafs of $\mathsf{T}_{R_{rev}}$ when the suffix tree is constructed.

As an example let R and R_{rev} be as follows:

$R =$	c	d	a	b	c	d	a	a	b	a	b	c	d	e
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
R_{rev}	e	d	c	b	a	b	a	a	d	c	b	a	d	c
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	15	14	13	12	11	10	9	8	7	6	5	4	3	2

We have numbered the suffixes of R and R_{rev} and for R_{rev} the values of the indices in $I'_{R_{rev}}$ are also listed. Assume that we want to answer a substring concatenation query on the pair $(8, 9)$ and $(1, 3)$ that corresponds to substrings `ab` and `cda` in R. Then we have that $I(v_f) = \{1, 5\}$ and $I'(v_b) = \{12, 10, 5\}$ and in this case we find that the two sets have the common index 5. This means that substring `abcda` starting at index $3 = 5 - |\texttt{ab}|$ of R is an answer to the substring concatenation query. Amir et al. [ALLS07] use *weighted ancestor queries*, introduced by Farach and Muthukrishnan [FM96], to locate v_f and v_b fast. A weighted ancestor query generalizes a predecessor query for trees such that we for each node can answer a predecessor query among its ancestors. For this we need to associate strictly increasing integer weight to the nodes on the path from the root to a node. In suffix trees, this weight is simply the string depth of each node. They improve the solution by Farach and Muthukrishnan, by introducing a deterministic data structure that uses $O(r)$ construction time and space, and answers queries in $O(\log \log r)$ time, where r is the number of nodes in the tree. The solution of Farach and Muthukrishnan have the same complexities but the construction time and space bound are only given with high probability. Further, they use a technique by Buchsbaum et al. [BGW00] to construct a data structure in $O(r\sqrt{\log r})$ time and space for deciding if any leafs in the substrees of v_f and v_b contains a common index in $O(\log \log r)$ time. In total, they use $O(r\sqrt{\log r})$ space and time to construct a data structure that answers substring concatenation queries in $O(\log \log r)$ time.

The Scheme by Gawrychowski et al. Gawrychowski et al. [GLN14] give an improved data structure for answering weighted ancestor queries on suffix trees in constant time and describe how it can be used for answering substring concatenation queries faster than the scheme by Amir et al. [ALLS07]. They observe that a weighted ancestor query is the running time bottleneck for answering *unrooted longest common prefix* (LCP) queries in a scheme by Cole et al. [CGL04]. An unrooted LCP query in a suffix tree T_S takes a node v of the suffix tree and an index i of S, and gives a node v' that is a descendant of v in T_S such that the concatenation of the labels on the path between v and v' has the longest common prefix with the suffix i of S over all descendants of v. A substring concatenation query on the indices (i, i'),(j, j') can then be answered as follows: We perform a weighted ancestor query in the suffix tree T_R on the leaf that represents suffix i to find the node v that represents prefix $i' - i$ of suffix i. This corresponds to matching $R[i, i']$ in T_R. From node v we perform a unrooted LCP query and obtain node v'. If the string depth of v' corresponds to the length of $R[i, i']R[j, j']$ then the path from v to v' spells out $R[j, j']$ and therefore $R[i, i']R[j, j']$ occurs as the prefix of every suffix represented by the leafs in the subtree of v'. The weighted ancestor data structure by Gawrychowski et al. [GLN14] answers queries in constant time an $O(r)$ space. The data structure for answering unrooted LCP queries by Cole et al. [CGL04] uses $O(r \log r)$ construction time and space and the query time in suffix trees is improved from $O(\log \log r)$ to $O(1)$ by the fast weighted ancestor data structure. In total, Gawrychowski et al. [GLN14] answer substring concatenation queries on R in constant time and $O(r \log r)$ construction time and space.

Our solution The following is an extended description of our solution that leads to branch (i) of Theorem 3. As the basis for our solution we use a suffix tree over R, T_R, that we augment with the weighted ancestor data structure of Gawrychowski et al. [GLN14] and a *lowest common ancestor* (LCA) data structure by Harel and Tarjan [HT84]. The LCA data structure enables us to answer longest common prefix queries on any suffix pair of R. We need the following definitions from Section 2.4: We let $x = R[i, i']$ and $y = R[j, j']$ and we let v_x and v_y be the nodes obtained

by matching x and y in T_R. For a substring x of R, let $S(x)$ denote the suffixes of R that have x as a prefix, and let $S'(x) = \{i + |x| \mid i \in S(x) \wedge i + |x| \leq n\}$, i.e., $S'(x)$ are the suffixes of R that are immediately preceded by x. Hence, for two substrings x and y, the suffixes that have xy as a prefix are exactly $S'(x) \cap S(y)$. Let $\text{rank}(i)$ be the position of suffix $R[i..r]$ in the lexicographic ordering of all suffixes of R, and let $\text{rank}(A) = \{\text{rank}(i) \mid i \in A\}$ for $A \subseteq \{1, 2, \ldots, n\}$. Then xy is a substring of R if and only if $\text{rank}(S'(x)) \cap \text{rank}(S(y)) \neq \emptyset$. The set of suffixes of R with y as a prefix will occupy a consecutive range of the lexicographic ordering. This means we can represent the set $\text{rank}(S(y))$ as a range $[y_{min}, y_{max}]$, where $y_{min} = \min(\text{rank}(S(y)))$ and $y_{max} = \max(\text{rank}(S(y)))$, and the intersection problem can then be formulated as *one dimensional range emptiness query*: Report any element of $\text{rank}(S'(x))$ that is within the range $[y_{min}, y_{max}]$. Ideally, we would like to store the set $\text{rank}(S'(x))$ in each node v_x of T_R and answer queries as follows: First, we perform a weighted ancestor query on x that gives us the node v_x. We then find the range $[y_{min}, y_{max}]$ with a weighted ancestor query on y that gives us the node v_y. Then the rank of the leftmost and rightmost leaf of the subtree spanned by v_y corresponds to y_{min} and y_{max}. We then perform a 1D range emptiness query by checking if any element of $\text{rank}(S'(x))$ is in the range $[y_{min}, y_{max}]$. We use a result by Belazzougui et al. [BBPV10] for answering 1D range emptiness queries that stores a set $A \subseteq [1, r]$, using $O(|A| \log^{\epsilon} r)$ *bits* of space, for any constant $\epsilon > 0$, and answers queries in constant time. If we use this data structure for every node in T_R then the total number of elements stored is, in the worst case, $\Omega(r^2)$ such that we use use $\Omega(r^2 \log^{\epsilon} r)$ bits of space in total for all the range emptiness data structures. Instead, we employ a *heavy path decomposition* [HT84] of T_R and only store a range emptiness data structure in the top node of every heavy path. Each suffix can then only be present in $O(\log r)$ range emptiness data structures and this reduces the space to $O(r \log^{1+\epsilon} r)$ *bits* equivalent to $O(r \log^{\epsilon} r)$ words. We need to perform our queries slightly different when we have less range emptiness data structures. We locate v_x as before and if v_x is a node at the top of a heavy path then we locate v_y and perform the range emptiness query on the set $\text{rank}(S'(x))$ with the range $[y_{min}, y_{max}]$. Otherwise, we need to decide how far y follows the heavy path from v_x. Let l be the leaf where the heavy path ends and let h be the index of the suffix represented by l. We can decide how long y follows the heavy path with a longest common prefix query between a suffix j (remember $y = R[j, j']$) and suffix $h + |x|$ of R. If all of y is matched on the heavy path, then we have reached a node where every leaf in the subtree represents a suffix that is prefixed by xy and we can report index h to answer the substring concatenation query. Otherwise, y at some point deviates from the heavy path below v_x and we end up in a node $v_{x'}$ that is the top node of a new heavy path. Let y' be the remaining unmatched suffix of y such that $xy = x'y'$. Finding an occurrence of $x'y'$ corresponds to finding an occurrence of xy. We find the range $[y'_{min}, y'_{max}]$ with a weighted ancestor query and perform a range emptiness query on the set $\text{rank}(S'(x'))$ that is stored in $v_{x'}$. We perform a constant number of constant time operations such that we answer substring concatenation queries in constant time.

1.2 Subsequence Automata with Default Transitions

In Chapter 3 we study automata for recognizing *subsequences* of strings. Given a string S of length n, a subsequence of S is any string obtained by deleting zero or more characters from S. Hence, if $S = s_1 s_2 \ldots s_n$ then a subsequence of S is any string $S' = s_{i_1} s_{i_2} \ldots s_{i_k}$ such that $i_j < i_{j+1}$ for all $1 \leq j < k$. An automata based approach for recognizing subsequences was studied by Baeza-Yates [BY91] and several other researches have subsequently studied the automaton and its

variations [TS05,CMT03,CT99,Tro01a,HSTA00,FFMR10,BIST03,Tro99]. The autmaton by Baeza-Yates is in the literature known as the *directed acyclic subsequence graph* (DASG) but here we will denote it as the *subsequence automaton* (SA). The SA is a standard *deterministic finite automaton* (DFA) [Huf54, Mea55, Moo56] with $n+1$ states that we identify with the integers $\{0, 1, \ldots, n\}$. It accepts string P iff P is a subsequence of S. We let $\Sigma(T)$ denote the alphabet of the string T. Each state s in the automaton is accepting and have the following transitions:

- For each character α in $\Sigma(S[s+1, n])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.

The automaton is constructed such that the state s, representing index s of S, has a transition with label α to the state that represents the next index of S that contain an occurrence of character α. The size of the automaton is measured as the number of states and transitions. Each transition of the SA has at most σ transitions so the size is $O(n\sigma)$.

For DFAs, with an alphabet of size σ, we require that every state has σ outgoing transitions, each labeled with distinct characters from the alphabet. This restricts the flexibility of how we can structure the automaton for recognizing subsequences. Instead, we consider DFAs with *default transitions* such that states are allowed to have labeled transitions for only a subset of the alphabet and a single unlabeled default transition to handle every missing transitions. If the automaton in a given state is unable to transition on a labeled transition, because no transition is labeled with the next character of P, it will follow the default transition without consuming the next character of P. We say that the *size* of the automaton is the sum of the number of states and transitions and that the *delay* of the automaton is the maximum number of default transitions that can be followed in the automaton before consuming a character of P. In Chapter 3 we introduce *subsequence automata with default transitions* (SADs) and investigate what kind trade-offs between size and delay that are possible. Our main result is given in Theorem 7.

Theorem 7. *Let S be a string of n characters from an alphabet of size σ. For any integer parameter k, $1 < k \leq \sigma$, we can construct a subsequence automaton with default transitions of size $O(nk \log_k \sigma)$ and delay $O(\log_k \sigma)$.*

An interesting aspect of the SAD constructions are that they are based on the properties of the well studied *ruler function* [Slo]. For an integer s, the ruler function gives the exponent of the largest power of two that divides s. We use the ruler function for assigning a level to each state but refer to the ruler function as the *level function* because the graphical presentation of the SADs, e.g as given in Figure 3.2, has a nice layout when the states are displaced vertically according to their level. The level of state s is given as:

$$\text{level}(s) = \max(\{x \mid s \bmod 2^x = 0\})$$

The number of transitions out of a state is decided based on its level such that a state s has at most $2^{\text{level}(s)}$ transitions plus a default transition. The overall idea behind the constructions of the automata are that a default transition always leads to a state with a higher level. This means that a default transition leads to a state with at least twice the number of transitions such that the number of states the automaton skips when following a default transition increases exponentially. This guarantees that the delay is $O(\log n)$. We reduce this to $O(\log \sigma)$ by observing that a state at level $\log \sigma$ will have σ transitions such that the state has a transition for every character of the alphabet. Hence, a default transition will never be followed from states with level larger than $\log \sigma$.

At this point we have a subsequence automaton with default transitions of size $O(n \log n)$ and delay $O(\log n)$. We obtain the full trade-off between size and delay by introducing an integer k such that $1 < k \leq \sigma$, and changing the ruler function to give the exponent of the largest power of k that divides s. In this way, the number of transitions will increase with a factor k instead of a factor 2 when we follow a default transition.

We further study automata over multiple strings. Crochemore et al. [CT99, CMT03] generalizes the simple subsequence automaton to accept patterns that are subsequences of multiple strings. For a set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$ they define the *subsequence automaton for multiple strings* that accepts pattern P if P is a subsequence of at least one string in $\mathcal{S}$ and the *common subsequence automaton for multiple strings* that accepts pattern P if P is a subsequence of every string in $\mathcal{S}$. In the same way we generalize the subsequence automaton with default transitions for accepting subsequences of multiple strings. We only describe the construction of the automaton that accepts a pattern P if P is a subsequence of every string in $\mathcal{S}$ but the construction of the version follows immediately. The result obtained for subsequence automata with default transitions over multiple strings is given in Theorem 8 on page 43.

In Section 1.2.1 we use automata with default transitions for solving substring matching but we take another focus by making the automaton represent the pattern P in order to find occurrences of P in the text S. Section 1.2.2 briefly describes the known results for matching subsequences in the word RAM model.

1.2.1 Substring Matching

In Chapter 3 we exclusively considers automata that recognizes subsequences of S. The original idea of introducing default transitions to subsequence matching automata was inspired by the KMP algorithm by Knuth et al. [KJHMP77] for *substring* matching. The KMP algorithm preprocesses a pattern P to find exact occurrences of P in a text S. Standard textbooks are usually not presenting the KMP algorithm as an automaton, but the algorithm can be seen as simulating a deterministic automaton with default transitions: The automaton contains $m+1$ states where state s_i of the KMP automaton represent the prefix i of P such that the automaton will be in state s_i when a length i prefix of P has been found. For every $0 \leq i < m$ the state s_i has a transition with label $P[i+1]$ to state s_{i+1} and a default transition to the state $s_{i'}$ such that $s_{i'}$ is the state that represents the longest prefix of P that is a suffix of the prefix represented by s_i while at the same time $P[i'+1] \neq P[i+1]$. The default transition in the KMP automaton corresponds to the *next* array that Knuth et al. use in [KJHMP77] for storing the next shift of the pattern when a mismatch occurs. Generally more known is the standard DFA construction for substring matching. The principle of operation is the same as for the KMP automaton but on a mismatch, meaning that no progression is made towards a match of the pattern, the automaton transitions directly to the state that the KMP automaton would find by following default transitions. This means that each state of the standard string matching automaton has σ transitions. The size of the standard string matching DFA is $O(m\sigma)$ and the delay is $O(1)$. For the KMP automaton the size is $O(m)$ and Knuth et al. [KJHMP77] show that the delay is $O(\log m)$. As in our study of subsequence automata in Chapter 3 it is possible to construct substring matching automata that compromises between *one* and σ labeled transitions. Simon [Sim94, Han93] showed that at most $2m$ of the edges in the standard DFA does not go to the initial state. By introducing default transitions to the standard DFA, we can represent the transitions of each state that go to the initial state as a single default transition. Besides the $O(m)$ default transitions we only have at most $2m$ other transitions such that the size of this automaton

becomes $O(m)$. The delay becomes constant since we can never follow more than a single default transition before we consume a character from P.

1.2.2 The subsequence problem in the RAM model

A simple approach for solving the subsequence problem in the word RAM model is to traverse S from left to right while matching characters from P when they are encountered in S. If every character from P is matched then P is a subsequence of S. This scheme uses $O(n)$ time and constant space except for storing S and P.

For the data structure version of the problem, where S is preprocessed to get fast subsequent queries, Bille et al. [BFC08] showed a scheme that combines the subsequence automaton of Baeza-Yates [BY91] with a successor data structure. Both of these data structures can individually be used for subsequence searching and Section 1.2 already gave a brief description of the subsequence automaton. The successor data structure solution works as follows: For each character α, they store a successor data structure that contains every index with character α. With these successor structures it is possible to simulate the subsequence automaton such that each transition is simulated with a successor query. With successor data structures such as van Embde boas trees [vEBKZ77] or Y-fast tries [Wil83] they obtain a $O(m \log\log n)$ time solution for subsequence searching that uses $O(n)$ space. With a combination of the automaton and successor solutions, Bille and Farach-Colton [BFC08] reduce the $\log\log n$ factor to $\log\log \sigma$ in the following way: A subsequence automaton is constructed from the string S but only every σ-th state have outgoing transitions, i.e. only states $i\sigma$ for $0 \leq i \leq \lfloor n/\sigma \rfloor$ have outgoing transitions. For each state $i\sigma$ with outgoing transitions they build σ successor data structures such that successor structure D_α contains the indices from $i\sigma + 1$ to $(i+1)\sigma$ of S that contains character α. We search for subsequences as follows: Assuming that we have matched prefix $P[0..i]$ and the automaton is in state s, we match character $\alpha = P[i+1]$ as follows: If state s has outgoing transitions and specifically an outgoing transitions with label α, we follow the transition out of s to state s' and have now matched prefix $P[0..i+1]$. If no such transition exists we conclude that P is not a subsequence of S. If state s does not have outgoing transition, we query the successor data structure D_α of the largest state $s^* < s$ that has outgoing transitions (i.e $s^* = \sigma\lfloor s/\sigma \rfloor$) with value s. If the successor query returns index s', we move to state s' and have now matched prefix $P[0..i+1]$. If no index in D_α is larger than s, we move to the smallest state $s^* > s$ that have outgoing transitions and continue as in the first case above. If we are not able to find $s^* > s$ then we conclude that P is not a subsequence of S. Bille and Farach-Colton [BFC08] spend $O(\log\log n)$ time for each character matched such that they decide if P is a subsequence of S in $O(m \log\log \sigma)$ time. For each state with outgoing transitions they use $O(\sigma)$ space but because they only have $O(n/\sigma)$ of these they use $O(n)$ space in total. In total Bille and Farach-Colton [BFC08] spend $O(n)$ time constructing all states and $O(n)$ time constructing all successor data structures.

1.3 Deterministic Indexing for Packed Strings

In Chapter 4 we study substring matching in word packed strings. Given a string $S = s_1 s_2 \dots s_n$, a substring of S is any string $S' = s_i s_{i+1} \dots s_{i+k}$ such that $1 \leq i \leq i+k \leq n$. We are interested in preprocessing S into a data structure such that we afterwards can decide fast if a string P is a substring of S. The two classic solutions to this problem are *suffix trees* [Wei73] and *suffix*

arrays [MM93]. The suffix tree of S is a compacted trie over all suffixes of S and the suffix array is an array over the indices of S sorted according to the lexicographic ordering of the suffixes they represent. Section 4.2 of Chapter 4 defines suffix trees and suffix arrays in more detail.

In their basic form, both suffix trees and suffix arrays of S are able answer the following queries on P:

$\mathsf{Count}(P)$: Return the number of occurrence of P in S.

$\mathsf{Locate}(P)$: Report all occurrences of P in S.

$\mathsf{Predecessor}(P)$: Returns the predecessor of P in S, i.e., the lexicographic largest suffix in S that is smaller than P.

In Chapter 4 we are interested in constructing a data structure for answering the above queries in a setting where all strings are *word packed*, all construction times; query times and space bounds are deterministic and where we only use $O(n)$ space. On a standard unit-cost word RAM with word size $w = \Theta(\log n)$ we obtain the following result:

Theorem 9. *Let S be a string of length n over an alphabet of size σ and let $\alpha = w/\log\sigma$ be the number of characters packed in a word. Given S we can build an index in $O(n)$ deterministic time and space such that given a packed pattern string of length m we can support* Count *and* Predecessor *in time $O(\frac{m}{\alpha} + \log m + \log\log\sigma)$ and* Locate *in time $O(\frac{m}{\alpha} + \log m + \log\log\sigma + \mathrm{occ})$ time.*

Theorem 9 hold under the assumption that the alphabet size σ is polynomial in n. Otherwise, we need additional time for sorting the alphabet [FCFM00].

Our result work on word packed strings that we define in the next section. We then explain how our result is related to previous results by Cole et al. [CKL06] and Fischer and Gawrychowski [FG15]. Lastly, we explain how we solve the lexicographic predecessor problem for short strings.

1.3.1 Word Packed Strings

For a string we usually store each character in a word of memory, but if the characters are taken from an alphabet of size σ, then we only need $\lceil\log\sigma\rceil$ bits for representing each character. When we process a string S of length n in the word RAM model, we assume that each memory word consist of $w \geq \log n$ bits such that we can address every index of S in constant time. In cases where σ is much smaller than n, such that $w > \lceil\log\sigma\rceil$, we are wasting $w - \lceil\log\sigma\rceil$ bits of memory for each character we store. With word packed strings we store $\alpha = \lfloor w/\lceil\log\sigma\rceil\rfloor$ characters in a single word by concatenating together the bits that represents α consecutive characters of S. Let S' be the array that stores the word packed version of S. Then we have $S[i] = (S'[\lfloor i/\alpha\rfloor] << u) >> d$, where $u = \lceil\log\sigma\rceil(\alpha - (i \bmod \alpha) - 1)$, $d = \lceil\log\sigma\rceil(\alpha - 1)$ when S and S' are zero indexed. The integer division in the index finds the correct word of S' and the shifts are masking the irrelevant part of the word. In Chapter 4 we implicit use this indexing scheme for word packed strings.

1.3.2 Combining Suffix Trees and Suffix Arrays

As mentioned earlier, we are interested in the setting where construction times, query times and space bounds are deterministic and where we only use $O(n)$ space. For the suffix tree, this rule out the use of randomized hashing schemes for navigation where either construction time or query is expected.

A simple and deterministic solution for navigating a node in a suffix tree is to store an array with pointers to each child. Let c be number of children of the node. We can then either have c-sized array that can be navigated in $O(\log c)$ time or a σ-sized array that can be navigated in constant time. Globally, with n nodes in the suffix tree, the first solution uses $O(n)$ space and the second uses $O(n \log \sigma)$ space. The queries listed above can be answered in $O(m \log \sigma)$ time and $O(m)$ time, respectively, where m is the length of P.

Queries in a suffix array over S corresponds to a binary search over the lexicographically ordered suffixes of S. This takes $O(m \log n)$ time because we might read $O(m)$ characters of P each time we half the search range. This can be reduced to $O(m + \log n)$ with the help of precomputed information on the *longest common prefix* between a selected number of suffixes of S.

The Suffix Tray by Cole et al. Our result is based on the *suffix tray* of Cole et al. [CKL06] that combine a suffix tree and a suffix array. Instead of settling for either the tree or the array, they construct a combination where a search in the top nodes of the suffix tree is used to reduce the range of the suffix array where matches can reside. With this technique they only need a small number of nodes from the suffix tree that can each have a σ-sized navigation array without using more than $O(n)$ space in total. They define *σ-nodes* to be the nodes of the suffix tree that span at least σ of the leafs. We will call σ-nodes for *heavy* nodes. Heavy nodes are further divided into *branching*, *non-branching* and *leafs*: A heavy branching node has at least two heavy nodes among its children, a heavy non-branching node has exactly one heavy child among its children and a heavy leafs have no heavy nodes among its children.

They use a σ-sized navigation array for the heavy branching nodes and a c-sized navigation array for both heavy non-branching and leaf nodes, but the non-branching nodes additionally have a pointer to its single heavy child. For every other node v that has a heavy node as its parent, they store two indices that indicate the range in the suffix array that contains the same suffixes as the subtree below v. We navigate the suffix tray as follows: From the root we consume characters of P as long as only heavy branching, non-branching nodes and leafs are encountered. Branching nodes are navigated with a constant time lookup in the navigation array. Non-branching nodes are navigated by first checking if we should follow the single pointer to the heavy child and if not, by performing a binary search on the c-sized array to find the correct child. Leafs are navigated by a binary search on the c-sized array. When we encounter any other node we search the suffix array in the range indicated by the two indices stored in the node. In the suffix tree we spend $O(m)$ time on the heavy branching and non-branching nodes and possibly $O(\log \sigma)$ time for the single binary search that leaves the heavy node. For the suffix array search we spend $O(m + \log \sigma)$ time because the range we search contains at most σ suffixes. In total we spend $O(m + \log \sigma + occ)$ answering queries with the suffix tray. Each heavy leaf spans at least σ leafs of the suffix tree and all heavy leafs spans disjoint subtrees such that we have at most n/σ heavy leafs. Since we have at most one heavy branching node per heavy leaf we also have at most n/σ heavy branching nodes. Each heavy branching node uses $O(\sigma)$ space such that we in total, for all heavy branching nodes, use $O(n)$ space. All other nodes use space proportional to its number of children such that we in total uses $O(n)$ space for the suffix tree. The suffix array uses $O(n)$ space and in total we use $O(n)$ space for the suffix tray.

The Suffix Tray with Deterministic Hashing Fischer and Gawrychowski [FG15] realized that they could improve the query times to $O(m + \log \log \sigma + occ)$ by using deterministic perfect hashing

for navigating the heavy branching nodes. They use the same idea as Cole et al. but instead of using a σ-sized array for the heavy branching nodes they use a deterministic hashing scheme by Ružić [Ruž08, Theorem 3]. They increase the number of heavy branching nodes by allowing each to span only $\log^2 \log \sigma$ leafs while still obtaining constant time navigation and $O(n)$ total space for the suffix tree. In Chapter 4 we extend the result by Fischer and Gawrychowski [FG15] to efficiently work with word packed strings. We do this by using deterministic hashing for navigating the suffix tree in chunks of α characters and extending the standard suffix array matching algorithm to run in $O(m/\alpha + \log n)$ time for word packed strings.

1.3.3 The Lexicographic Predecessor Problem for Short Strings

Given a pattern P and a set of strings $\mathcal{S}$, the lexicographic predecessor problem is to find the lexicographically largest string in $\mathcal{S}$ that is also lexicographically smaller than P. In Chapter 4 we specifically solve the lexicographic predecessor problem for small strings of at most α characters that each fit in a word of memory. We claim that we can use a standard predecessor data structure over the integers for solving the lexicographic predecessor problem by treating each string as an integer. Here we rigorously show why this work. We assume that each short string is stored in a word such that the $\lceil \log \sigma \rceil$ most significant bits represent the first character of the string and that the lexicographic order of the characters in the alphabet corresponds to the order of the numeric value of the bits used for representing the characters. If a string is less than α characters long we extend it with the special character \$ such that every string is of length α, where character \$ is a special character that is lexicographically smaller than any other character from the alphabet. We insert every string from $\mathcal{S}$ into a predecessor data structure D. Given a string P of length α, stored in a word, we show that a predecessor query on P in D gives the lexicographic predecessor of P. Assume that the query returns the value $P' < P$ and let k be the most significant bit position of the words where P' and P has a `0` and `1`, respectively. Let bit position k be part of the bits that represent character position i of both P' and P. This means $P'[i]$ is lexicographically smaller than $P[i]$ and since the values agree on every bit with higher significance than the bit at position k, we conclude that P' is lexicographically smaller than P. We still need to show that P' is the lexicographically largest string that is smaller than P. For contradiction, assume that a string P'' exists in $\mathcal{S}$ such that $P' <_{lex} P'' <_{lex} P$ and let k' be the most significant bit position where P' and P'' disagree. Since $P' <_{lex} P''$ this means that the bits at position k' of P' and P'' are `0` and `1`, respectively, which makes $P' < P''$ and contradict the fact that the predecessor query of P returned P'. From this we conclude that P' is the lexicographic predecessor of P.

1.4 Dynamic Partial Sums in Constant Time and Succinct Space with the Ultra Wide Word RAM Model

In Chapter 5 we study *dynamic partial sums* in the *ultra wide word RAM model*. The dynamic partial sums problem is to maintain an integer array A of n elements under the following operations:

$\mathsf{Access}(i)$: Return $A[i]$.

$\mathsf{Sum}(i)$: Returns the sum of every index smaller than or equal to i, i.e. $\sum_{j=1}^{i} A[i]$.

$\mathsf{Update}(i, v)$: Increment index i of A with value v, i.e $A[i] := A[i] + v$, where v is any value representable in two's compliment representation with w-bits.

In this chapter we consider a different variation of the dynamic partial sums problem than the one we discuss in Chapter 2. Specifically, we do not support the $\mathsf{search}(i)$ operation and any of the operations that change the length of A. We solve the problem on the ultra wide word RAM model (UW-RAM) that was recently introduced by Farzan et al. [FLONS15]. The UW-RAM is a standard unit-cost word RAM with a word size $w \geq \log n$ and a standard instruction set but it extends the model by enabling instructions that work on *wide words* and parallel access to memory. More specifically, every arithmetic and logical instruction can work on wide words containing w^2 bits in constant time. This makes word-level parallelism very attractive in this model. The model also supports reading w words from memory in constant time. This can either be w consecutive words or w scattered words, where the addresses of each word is given as w consecutive bits of a wide word.

Farzan et al. [FLONS15] show that the UW-RAM can simulate the RAMBO model with a space overhead and in this way obtain a result for the dynamic partial sums problem directly from a result by Brodnik et al. [BKMN06]. The RAMBO model was suggested by Fredman and Saks [FS89] and introduced by Brodnik [Bro95]. The general idea is to overcome lower bounds by constructing a memory model where bits of memory can be shared between memory words.

We use the *fenwick tree* [Fen94] as data structure for representing the array A. The fenwick tree supports Access, Sum and Update operations in $O(\log n)$ time on the standard unit-cost word RAM and can be constructed in $O(n)$ time. We use the same memory layout in the UW-RAM but show how the operations can be sped up with the use of the strong primitives for parallel memory access and manipulation of w^2-bit words. The $O(\log n)$ time bound in the word RAM model comes from accessing $O(\log n)$ location. We show how to calculate and access these memory locations in constant time on the UW-RAM and obtain the following result:

Theorem 10. *Let A be an array of n positive w-bit integers. In the ultra wide word-RAM model with multiplication we can build a data structure in $O(n)$ time and $n + O(\log n)$ words of space that support Access, Sum and Update operations in $O(1)$ time.*

We also consider the problem in a UW-RAM *without* support for multiplication and show the following result:

Lemma 9. *Let A be an array of n positive w-bit integers. In the ultra wide word-RAM model without multiplication we can build a data structure in $O(n)$ time and $n + O(\log n)$ words of space that support Access, Sum and Update operations in $O(\log \log n)$ time.*

The $O(\log n)$ words of space comes from storing constants of size $O(\log^2 n)$ bits which we use for implementing word-level parallelism. In the unit-cost word RAM model the fenwick tree also support the search operation (often called select in the literature) in $O(\log n)$ time. The $\mathsf{search}(v)$ operation returns index i of A where $\mathsf{Sum}(i-1) < v \leq \mathsf{Sum}(i)$. It seems difficult to speed up this operation in UW-RAM. We pose as an open problem whether it is possible to maintain an array of n w-bit integers on the UW-RAM while supporting Access, Sum, Update and Select operations in constant time.

Chapter 2

Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation

Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation

Philip Bille Patrick Hagge Cording Inge Li Gørtz Frederik Rye Skjoldjensen
Hjalte Wedel Vildhøj Søren Vind

Technical University of Denmark

Abstract

Given a static reference string R and a source string S, a relative compression of S with respect to R is an encoding of S as a sequence of references to substrings of R. Relative compression schemes are a classic model of compression and have recently proved very successful for compressing highly-repetitive massive data sets such as genomes and web-data. We initiate the study of relative compression in a dynamic setting where the compressed source string S is subject to edit operations. The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. We present new data structures that achieve optimal time for updates and queries while using space linear in the size of the optimal relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates. To achieve these results, we revisit the dynamic partial sums problem and the substring concatenation problem. We present new optimal or near optimal bounds for these problems. Plugging in our new results we also immediately obtain new bounds for the string indexing for patterns with wildcards problem and the dynamic text and static pattern matching problem.

2.1 Introduction

Given a static reference string R and a source string S, a *relative compression of S with respect to R* is an encoding of S as a sequence of references to substrings of R. Relative compression (or *external macro compression*) is a classic model of compression defined by Storer and Szymanski [SS78, SS82] in 1978 and has since been used in a wide range of compression scenarios [LDK99, LDK$^+$98, KPZ10, KPZ11, COM$^+$12, DJSS14, HPZ11]. To compress massive highly-repetitive data sets, such as biological sequences and web collections, relative compression has been shown to be very practical [KPZ10, KPZ11, HPZ11].

Relative compression is often applied to compress multiple similar source strings. In such settings relative compression is superior to compressing the source strings individually. For instance, human genomes are 99% similar and hence relative compression might be used to compress a large collection of sequenced genomes using, e.g., the human reference genome as the static reference string. We focus on the case of compressing a single source string, but our results trivially generalize to compressing multiple source strings.

In this paper we initiate the study of relative compression in a *dynamic setting*, where the compressed source string S is subject to edit operations (insertions, deletions, and replacements

of single characters). The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. Efficient data structures supporting these operations allow us to avoid costly recompression of massive data sets after updates.

We provide the first non-trivial bounds for this problem. We present new data structures that achieve *optimal* time for updates and queries while using space linear in the size of the *optimal* relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates.

To achieve these results, we revisit the *dynamic partial sums problem* and the *substring concatenation problem*. We present new optimal or near optimal bounds for both of these problems (see detailed discussion below). Furthermore, plugging in our new results immediately leads to new bounds for the *string indexing for patterns with wildcards problem* [LNV14, BGVV14] and the *the dynamic text and static pattern matching problem* [ALLS07].

2.1.1 Dynamic Relative Compression

Given a *reference string* R and a *source string* S, a *relative compression of S with respect to R* is a sequence $C = (i_1, j_1), ..., (i_{|C|}, j_{|C|})$ such that $S = R[i_1, j_1] \cdots R[i_{|C|}, j_{|C|}]$. We call C a *substring cover* for S. The substring cover is *optimal* if $|C|$ is minimum over all relative compressions of S with respect to R. The *dynamic relative compression problem* is to maintain a relative compression of S under the following operations. Let i be a position in S and α be a character.

$\mathsf{access}(i)$**:** return the character $S[i]$,

$\mathsf{replace}(i, \alpha)$**:** change $S[i]$ to character α,

$\mathsf{insert}(i, \alpha)$**:** insert character α before position i in S,

$\mathsf{delete}(i)$**:** delete the character at position i in S.

Note that operations insert and delete change the length of S by a single character. In all bounds below, the $\mathsf{access}(i)$ operation extends to decompressing an arbitrary substring of length ℓ using only $O(\ell)$ additional time.

Our Results Throughout the paper, let r be the length of the reference string R, N be the length of the (uncompressed) string S, and n be the size of an optimal relative compression of S with regards to R. All of the bounds mentioned below and presented in this paper hold for a standard unit-cost RAM with w-bit words with standard arithmetic and logical operations on a word. This means that the algorithms can be implemented directly in standard imperative programming languages such as C [KR78] or C++ [Str00]. An index into R or S can be stored in a single word and hence $w \geq \log(n + r)$.

Theorem 1. *Let R and S be a reference and source string of lengths r and N, respectively, and let n be the length of the optimal substring cover of S by R. Then, we can solve the dynamic relative compression problem supporting* access, replace, insert, *and* delete

(i) in $O(n + r)$ space and $O\left(\frac{\log n}{\log \log n} + \log \log r\right)$ time per operation, or

(ii) in $O(n + r \log^{\epsilon} r)$ *space and* $O\left(\frac{\log n}{\log \log n}\right)$ *time per operation, for any constant* $\epsilon > 0$.

These are the first non-trivial bounds for the problem. Together, the bounds are optimal for most natural parameter combinations. In particular, any data structure for a string of length N supporting access, insert, and delete must use $\Omega(\log N / \log \log N)$ time in the worst-case regardless of the space [FS89] (this is called the *list representation problem*). Since $n \leq N$, we can view $O(\log n / \log \log n)$ as a compressed version of the optimal time bound that is always $O(\log N / \log \log N)$ and better when S is compressible. Hence, Theorem 1(i) provides a linear-space solution that achieves the compressed time bound except for an $O(\log \log r)$ additive term. Note that whenever $n \geq (\log r)^{\log^{\epsilon} \log r}$, for any $\epsilon > 0$, the $\log n / \log \log n$ term dominates the query time and we match the compressed time bound. Hence, Theorem 1(i) is only suboptimal in the special case when n is almost exponentially smaller than r. In this case, we can use Theorem 1(ii) which always provides a solution achieving the compressed time bound at the cost of increasing the space to $O(n + r \log^{\epsilon} r)$.

We note that dynamic compression under different models of compression has been studied extensively [GGV03, FMMN04, FM05, SG06, FV07, JSS12, NN13]. However, all of these results require space dependent on the size of the original string and hence cannot take full advantage of highly-repetitive data.

2.1.2 Dynamic Partial Sums

The *partial sums problem* is to maintain an array $Z[1..s]$ under the following operations.

- sum(i)**:** return $\sum_{j=1}^{i} Z[j]$,
- update(i, Δ)**:** set $Z[i] = Z[i] + \Delta$,
- search(t)**:** return $1 \leq i \leq s$ such that sum$(i-1) < t \leq$ sum(i). To ensure well-defined answers, we require that $Z[i] \geq 0$ for all i.

The partial sums problem is a classic and well-studied problem [Die89, RRR01, HR03, FS89, HSS11, HRS96, Fen94, PD04]. In our context, we consider the problem in the word RAM model, where each array entry stores a w-bit integer and the element of the array can be changed by δ-bit integers, i.e., the argument Δ can be stored in δ bits. In this setting, Pătraşcu and Demaine [PD04] gave a linear-space data structure with $\Theta(\log s / \log(w/\delta))$ time per operation. They also gave a matching lower bound.

We consider the following generalization supporting dynamic changes to the array. The *dynamic partial sums problems* is to additionally support the following operations.

- insert(i, Δ)**:** insert a new entry in Z with value Δ before $Z[i]$,
- delete(i)**:** delete the entry $Z[i]$ of value at most Δ.
- merge(i)**:** replace entry $Z[i]$ and $Z[i+1]$ with a new entry with value $Z[i] + Z[i+1]$.
- divide(i, t)**:** , where $0 \leq t \leq Z[i]$. Replace entry $Z[i]$ by two new consecutive entries with value t and $Z[i] - t$, respectively.

Hon et al. [HSS11] and Navarro and Sadakane [NS14] presented optimal solutions for this problem in the case where the entries in Z are at most polylogarithmic in s (they did not explicitly consider the merge and divide operation).

Our Results We show the following improved result.

Theorem 2. *Given an array of length s storing w-bit integers and parameter δ, such that $\Delta < 2^\delta$, we can solve the dynamic partial sums problem supporting* sum, update, search, insert, delete, merge, *and* divide *in linear space and $O(\log s / \log(w/\delta))$ time per operation.*

Note that this bound simultaneously matches the optimal time bound for the standard partial sums problem and supports storing arbitrary w-bit values in the entries of the array, i.e., the values we can handle in optimal time are exponentially larger than in the previous results.

To achieve our bounds we extend the static solution by Pătraşcu and Demaine [PD04]. Their solution is based on storing a sampled subset of *representative elements* of the array and difference encode the remaining elements. They pack multiple difference encoded elements in words and then apply word-level parallelism to speedup the operations. To support insert and delete the main challenge is to maintain the representative elements that now dynamically move within the array. We show how to efficiently do this by combining a new representation of representative elements with a recent result by Pătraşcu and Thorup [PT14]. Along the way we also slightly simplify the original construction by Pătraşcu and Demaine [PD04].

2.1.3 Substring Concatenation

Let R be a string of length r. A *substring concatenation query* on R takes two pairs of indices (i, j) and (i', j') and returns the start position in R of an occurrence of $R[i, j]R[i', j']$, or `NO` if the string is not a substring of R. The *substring concatenation problem* is to preprocess R into a data structure that supports substring concatenation queries.

Amir et al. [ALLS07] gave a solution using $O(r\sqrt{\log r})$ space with query time $O(\log \log r)$, and recently Gawrychowski et al. [GLN14] showed how to solve the problem in $O(r \log r)$ space and $O(1)$ time.

Our Results We give the following improved bounds.

Theorem 3. *Given a string R of length r, the substring concatenation problem can be solved in either*

(i) $O(r \log^\epsilon r)$ space and $O(1)$ time, for any constant $\epsilon > 0$, or

(ii) $O(r)$ space and $O(\log \log r)$ time.

Hence, Theorem 3(i) matches the previous $O(1)$ time bound while reducing the space from $O(r \log r)$ to $O(r \log^\epsilon r)$ and Theorem 3(ii) achieves linear space while using $O(\log \log r)$ time. Plugging in the two solutions into our solution for dynamic relative compression leads to the two branches of Theorem 1.

To achieve the bound in (i), the main idea is a new construction that efficiently combines compact data structure for 1D range reporting [BBPV10] with the recent constant time weighted level ancestor data structure for suffix trees [GLN14]. The bound in (ii) follows as a simple implication of another recent result for *unrooted LCP queries* [BGVV14] by some of the authors. The substring concatenation problem is a key component in several solutions to the *string indexing for patterns with wildcards problem* [BGVV14, CGL04, LNV14], where the goal is to preprocess a string T to support pattern matching queries for patterns with wildcards. Plugging in Theorem 3(i) we immediately obtain the following new bound for the problem.

Corollary 1. *Let T be a string of length t. For any pattern string P of length p with k wildcards, we can support pattern matching queries on T using $O(t \log^{\epsilon} t)$ space and $O(p + \sigma^k)$ time for any constant $\epsilon > 0$.*

This improves the running time of fastest linear space solution by a factor $\log \log t$ at the cost of increasing the space slightly by a factor $\log^{\epsilon} t$. See [LNV14] for detailed overview of the known results.

2.1.4 Extensions

Finally, we present two extensions of the dynamic relative compression problem.

2.1.4.1 Dynamic Relative Compression with Access and Replace

If we restrict the operations to access and replace we obtain the following improved bound.

Theorem 4. *Let R and S be a reference and source string of lengths r and N, respectively, and let n be the length of the optimal substring cover of S by R. Then, we can solve the dynamic relative compression problem supporting* access *and* replace *in $O(n+r)$ space and $O(\log \log N)$ expected time.*

This version of dynamic relative compression is a key component in the *dynamic text and static pattern matching problem*, where the goal is to efficiently maintain a set of occurrences of a pattern P in a text T that is dynamically updated by changing individual characters. Let p and t denote the lengths of P and T, respectively. Amir et al. [ALLS07] gave a data structure using $O(t + p\sqrt{\log p})$ space which supports updates in $O(\log \log p)$ time. The computational bottleneck in the update operation is to update a substring cover of size $O(p)$. Plugging in the bounds from Theorem 4, we immediately obtain the following improved bound.

Corollary 2. *Given a pattern P and text T of lengths p and t, respectively, we can solve the dynamic text and static pattern matching problem in $O(t+p)$ space and $O(\log \log p)$ expected time per update.*

Hence, we match the previous time bound while improving the space to linear.

2.1.4.2 Dynamic Relative Compression with Split and Concatenate

We also consider maintaining a set of compressed strings under split and concatenate operations (as in Alstrup et al. [ABR00]). Let R be a reference string and let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be a set of strings compressed relative to R. In addition to access, replace, insert and delete we also define the following operations.

concat(i, j)**:** Add string $S_i \cdot S_j$ to $\mathcal{S}$ and remove S_i and S_j.

split(i, j)**:** Remove S_i from $\mathcal{S}$ and add $S_i[1, j-1]$ and $S_i[j, |S_i|]$.

We obtain the following bounds.

Theorem 5. *Let R be a reference string of length r, let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be a set of source strings of total length N, and let n be the total length of the optimal substring covers of the strings in $\mathcal{S}$. Then, we can solve the dynamic relative compression problem supporting* access, replace, insert, delete, split, *and* concat,

(i) in space $O(n+r)$ and time $O(\log n)$ for access *and time $O(\log n + \log\log r)$ for* replace, insert, delete, split, *and* concat, *or*

(ii) in space $O(n + r\log^{\epsilon} r)$ and time $O(\log n)$ for all operations.

Hence, compared to the bounds in Theorem 1 we only increase the time bounds by an additional $\log\log n$ factor.

2.2 Dynamic Relative Compression

In this section we show how Theorems 2 and 3 lead to Theorem 1. The proofs of Theorems 2 and 3 appear in Section 2.3 and Section 2.4, respectively.

Let $C = ((i_1, j_1), ..., (i_{|C|}, j_{|C|}))$ be the compressed representation of S. From now on, we refer to C as the *cover of* S, and call each element (i_l, j_l) in C a *block*. Recall that a block (i_l, j_l) refers to a substring $R[i_l, j_l]$ of R. A cover C is *maximal* if concatenating any two consecutive blocks $(i_l, j_l), (i_{l+1}, j_{l+1})$ in C yields a string that does not occur in R, i.e., the string $R[i_l, j_l]R[i_{l+1}, j_{l+1}]$ is not a substring of R. We need the following lemma.

Lemma 1. *If C_{MAX} is a maximal cover and C is an arbitrary cover of S, then $|C_{\text{MAX}}| \leq 2|C| - 1$.*

Proof. In each block b of C there can start at most two blocks in C_{MAX}, because otherwise two adjacent blocks in C_{MAX} would be entirely contained in the block b, contradicting the maximality of C_{MAX}. Since the last block of both C and C_{MAX} end at the last position of S, a contradiction of the maximality is already obtained when more than one block of C_{MAX} start in the last block of C. Hence, $|C_{\text{MAX}}| \leq 2|C| - 1$. □

Recall that n is the size of an optimal cover of S with regards to R. The lemma implies that we can maintain a compression of size at most $2n-1$ by maintaining a maximal cover of S. The remainder of this section describes our data structure for maintaining and accessing such a cover.

Initially, we can use the suffix tree of R to construct a maximal cover of S in $O(N + r)$ time by greedily matching the maximal prefix of the remaining part of S with any suffix of R. This guarantees that the blocks constitute a maximal cover of S.

2.2.1 Data Structure

The high level idea for supporting the operations on S is to store the sequence of block lengths $j_1 - i_1 + 1, \ldots, j_{|C|} - i_{|C|} + 1$ in a dynamic partial sums data structure. This allows us, for example, to identify the block that encodes the k^{th} character in S by performing a $\mathsf{search}(k)$ query.

Updates to S are implemented by splitting a block in C. This may break the maximality property so we use substring concatenation queries on R to detect if blocks can be merged. We only need a constant number of substring concatenation queries to restore maximality. To maintain the correct sequence of block lengths we use update, divide and merge operations on the dynamic partial sums data structure.

Our data structure consist of the string R, a substring concatenation data structure of Theorem 3 for R, a maximal cover C for S stored in a doubly linked list, and the dynamic partial sums data structure of Theorem 2 storing the block lengths of C. We also store auxiliary links between a block in the doubly linked list and the corresponding block length in the partial sums data structure, and

a list of alphabet symbols in R with the location of an occurrence for each symbol. By Lemma 1 and since C is maximal we have $|C| \leq 2n - 1 = O(n)$. Hence, the total space for C and the partial sums data structure is $O(n)$. The space for R is $O(r)$ and the space for substring concatenation data structure is either $O(r)$ or $O(r \log^{\epsilon} r)$ depending on the choice in Lemma 3. Hence, in total we use either $O(n + r)$ or $O(n + r \log^{\epsilon} r)$ space.

2.2.2 Answering Queries

To answer $\mathsf{access}(i)$ queries we first compute $\mathsf{search}(i)$ in the dynamic partial sums structure to identify the block $b_l = (i_l, j_l)$ containing position i in S. The local index in $R[i_l, j_l]$ of the i^{th} character in S is $\ell = i - \mathsf{sum}(l-1)$, and thus the answer to the query is the character $R[i_l + \ell - 1]$.

We perform $\mathsf{replace}$ and delete by first identifying $b_l = (i_l, j_l)$ and ℓ as above. Then we partition b_l into three new blocks $b_l^1 = (i_l, i_l + \ell - 2)$, $b_l^2 = (i_l + \ell - 1, i_l + \ell - 1)$, $b_l^3 = (i_l + \ell, j_l)$ where b_l^2 is the single character block for index i in S that we must change. In $\mathsf{replace}$ we change b_l^2 to an index of an occurrence in R of the new character (which we can find from the list of alphabet symbols), while we remove b_l^2 in delete. The new blocks and their neighbors, that is, b_{l-1}, b_l^1, b_l^2, b_l^3, and b_{l+1} may now be non-maximal. To restore maximality we perform substring concatenation queries on each consecutive pair of these 5 blocks, and replace non-maximal blocks with merged maximal blocks. All other blocks are still maximal, since the strings obtained by concatenating $b_{l'}$ with $b_{l'+1}$, for all $l' < l - 1$ and all $l' > l$, was not present in R before the change and is not present afterwards. A similar idea is used by Amir et al. [ALLS07]. We perform update, divide and merge operations to maintain the corresponding lengths in the dynamic partial sums data structure. The insert operation is similar, but inserts a new single character block between two parts of b_l before restoring maximality. Observe that using $\delta = O(1)$ bits in update is sufficient to maintain the correct block lengths.

In total, each operation requires a constant number of substring concatenation queries and dynamic partial sums operations; the latter having time complexity $O(\log n / \log(w/\delta)) = O(\log n / \log \log n)$ as $w \geq \log n$ and $\delta = O(1)$. Hence, the total time for each access, $\mathsf{replace}$, insert, and delete operation is either $O(\log n / \log \log n + \log \log r)$ or $O(\log n / \log \log n)$ depending on the substring concatenation data structure used. In summary, this proves Theorem 1.

2.3 Dynamic Partial Sums

In this section we prove Theorem 2. We support the operations $\mathsf{insert}(i, \Delta)$ and $\mathsf{delete}(i)$ on a sequence of w-bit integer keys by implementing them using update and a divide or merge operation, respectively. This means that we support inserting or deleting keys with value at most 2^{δ}.

We first solve the problem for small sequences. The general solution uses a standard reduction, storing Z at the leaves of a B-tree of large outdegree. We use the solution for small sequences to navigate in the internal nodes of the B-tree.

Dynamic Integer Sets We need the following recent result due to Pătraşcu and Thorup [PT14] on maintaining a set of integer keys X under insertions and deletions. The queries are as follows, where q is an integer. The membership query $\mathsf{member}(q)$ returns true if $q \in X$, predecessor $\mathsf{pred}_X(q)$ returns the largest key $x \in X$ where $x < q$, and successor $\mathsf{succ}_X(q)$ returns the smallest key $x \in X$

where $x \geq q$. The rank $\mathsf{rank}_X(q)$ returns the number of keys in X smaller than q, and $\mathsf{select}(i)$ returns the i^{th} smallest key in X.

Lemma 2 (Pătraşcu and Thorup [PT14])**.** *There is a data structure for maintaining a dynamic set of $w^{O(1)}$ w-bit integers that supports insert, delete, membership, predecessor, successor, rank and select in constant time per operation.*

2.3.1 Dynamic Partial Sums for Small Sequences

Let Z be a sequence of at most $B \leq w^{O(1)}$ integer keys. We will show how to store Z in linear space such that all dynamic partial sums operations can be performed in constant time. We let Y be the sequence of prefix sums of Z, defined such that each key $Y[i]$ is the sum of the first i keys in Z, i.e., $Y[i] = \sum_{j=1}^{i} Z[j]$. Observe that $\mathsf{sum}(i) = Y[i]$ and $\mathsf{search}(t)$ is the index of the successor of t in Y. Our goal is to store and maintain a representation of Y subject to the dynamic operations update, divide and merge in constant time per operation.

2.3.1.1 The Scheme by Pătraşcu and Demaine

We first review the solution to the static partial sums problem by Pătraşcu and Demaine [PD04], slightly simplified due to Lemma 2. Our dynamic solution builds on this.

The entire data structure is rebuilt every B operations as follows. We first partition Y greedily into *runs*. Two adjacent elements in Y are in the same run if their difference is at most $B2^{\delta}$, and we call the first element of each run a *representative* for all elements in the run. We use $\mathcal{R}$ to denote the sequence of representative values in Y and $\mathsf{rep}(i)$ to be the index of the representative for element $Y[i]$ among the elements in $\mathcal{R}$.

We store Y by splitting representatives and other elements into separate data structures: $\mathcal{I}$ and $\mathcal{R}$ store the representatives at the time of the last rebuild, while $\mathcal{U}$ stores each element in Y as an offset to its representative value as well as updates since the last rebuild. We ensure $Y[i] = \mathcal{R}[\mathsf{rep}(i)] + \mathcal{U}[i]$ for any i and can thus reconstruct the values of Y.

The representatives are stored as follows. $\mathcal{I}$ is the sequence of indices in Y of the representatives and $\mathcal{R}$ is the sequence of representative values in Y. Both $\mathcal{I}$ and $\mathcal{R}$ are stored using the data structure of Lemma 2. We can then define $\mathsf{rep}(i) = \mathsf{rank}_{\mathcal{I}}(\mathsf{pred}_{\mathcal{I}}(i))$ as the index of the representative for i among all representatives, and use $\mathcal{R}[\mathsf{rep}(i)] = \mathsf{select}_{\mathcal{R}}(\mathsf{rep}(i))$ to get the value of the representative for i.

We store in $\mathcal{U}$ the current difference from each element to its representative, $\mathcal{U}[i] = Y[i] - \mathcal{R}[\mathsf{rep}(i)]$ (i.e. updates between rebuilds are applied to $\mathcal{U}$). The idea is to pack $\mathcal{U}$ into a single word of B elements. Observe that $\mathsf{update}(i, \Delta)$ adds value Δ to all elements in Y with index at least i. We can support this operation in constant time by adding to $\mathcal{U}$ a word that encodes Δ for those elements. Since each difference between adjacent elements in a run is at most $B2^{\delta}$ and $|Y| = O(B)$, the maximum value in $\mathcal{U}$ after a rebuild is $O(B^2 2^{\delta})$. As B updates of size 2^{δ} may be applied before a rebuild, the changed value at each element due to updates is $O(B2^{\delta})$. So each element in $\mathcal{U}$ requires $O(\log B + \delta)$ bits (including an overflow bit per element). Thus, $\mathcal{U}$ requires $O(B(\log B + \delta))$ bits in total and can be packed in a single word for $B = O(\min\{w/\log w, w/\delta\})$.

Between rebuilds the stored representatives are potentially outdated because updates may have changed their values. However, observe that the values of two consecutive representatives differ by more than $B2^{\delta}$ at the time of a rebuild, so the gap between two representatives cannot be closed by B updates of δ bits each (before the structure is rebuilt again). Hence, an answer to $\mathsf{search}(t)$

cannot drift much from the values stored by the representatives; it can only be in a constant number of runs, namely those with a representative value $\mathsf{succ}_{\mathcal{R}}(t)$ and its two neighboring runs. In a run with representative value v, we find the smallest j (inside the run) such that $\mathcal{U}[j] + v - t > 0$. The smallest j found in all three runs is the answer to the $\mathsf{search}(t)$ query. Thus, by rebuilding periodically, we only need to check a constant number of runs when answering a $\mathsf{search}(t)$ query.

On this structure, Pătraşcu and Demaine [PD04] show that the operations sum, search and update can be supported in constant time each as follows:

$\mathsf{sum}(i)$**:** return the sum of $\mathcal{R}[\mathsf{rep}(i)]$ and $\mathcal{U}[i]$. This takes constant time as $\mathcal{U}[i]$ is a field in a word and representatives are stored using Lemma 2.

$\mathsf{search}(t)$**:** let $r_0 = \mathsf{rank}_{\mathcal{R}}(\mathsf{succ}_{\mathcal{R}}(t))$. We must find the smallest j such that $\mathcal{U}[j] + R[r] - t > 0$ for $r \in \{r_0 - 1, r_0, r_0 + 1\}$, where j is in run r. We do this for each r using standard word operations in constant time by adding $R[r] - t$ to all elements in $\mathcal{U}$, masking elements not in the run (outside indices $\mathsf{select}_{\mathcal{I}}(r)$ to $\mathsf{select}_{\mathcal{I}}(r+1) - 1$, and counting the number of negative elements.

$\mathsf{update}(i, \Delta)$**:** we do this in constant time by copying Δ to all fields $j \geq i$ by a multiplication and adding the result to $\mathcal{U}$.

To count the number of negative elements or find the least significant bit in a word in constant time, we use the technique by Fredman and Willard [FW93].

Notice that rebuilding the data structure every B operations takes $O(B)$ time, resulting in amortized constant time per operation. We can instead do this incrementally by a standard approach by Dietz [Die89], reducing the time per operation to worst case constant. The idea is to construct the new replacement data structure incrementally while using the old and complete data structure.

2.3.1.2 Efficient Support for divide and merge

We now show how to maintain the structure described above while supporting operations $\mathsf{divide}(i, t)$ and $\mathsf{merge}(i)$. An example supporting the following explanation is provided in Figure 2.1.

Observe that the operations are only local: Splitting $Z[i]$ into two parts or merging $Z[i]$ and $Z[i+1]$ does not influence the precomputed values in Y (besides adding/removing values for the divided/merged elements). We must update $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{U}$ to reflect these local changes accordingly. Because a divide or merge operation may create new representatives between rebuilds with values that do not fit in $\mathcal{U}$, we change $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{U}$ to reflect these new representatives by rebuilding the data structure locally. This is done as follows.

Consider the run representatives. Both $\mathsf{divide}(i, t)$ and $\mathsf{merge}(i)$ may require us to create a new run, combine two existing runs or remove a run. In any case, we can find a replacement representative for each run affected. As the operations are only local, the replacement is either a divided or merged element, or one of the neighbors of the replaced representative. Replacing representatives may cause both indices and values for the stored representatives to change. We use insertions and deletions on $\mathcal{R}$ to update representative values.

Since the new operations change the indices of the elements, these changes must also be reflected in $\mathcal{I}$. For example, a $\mathsf{merge}(i)$ operation decrements the indices of all elements with index larger than i compared to the indices stored at the time of the last rebuild We should in principle adjust the $O(B)$ changed indices stored in $\mathcal{I}$. The cost of adjusting the indices accordingly when using

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z	5	1	4	7	1	1	6	5	1	1	2	2	1	3	5	10	5	10	2
Y	5	6	10	17	18	19	25	30	31	32	34	36	37	40	45	55	60	70	72
$\mathcal{R}$	$\{5, 17, 25, 30, 45, 55, 60, 70\}$																		
$\mathcal{U}$	0	1	5	0	1	2	0	0	1	2	4	6	7	10	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	1	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	4	4	4	4	4	4	4	5	6	7	8	8

a) The initial data structure constructed from Z.

New index 9 Old index 9

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Z	5	1	4	7	1	1	6	3	2	1	1	2	2	1	3	5	10	5	10	2
Y	5	6	10	17	18	19	25	28	30	31	32	34	36	37	40	45	55	60	70	72
$\mathcal{R}$	$\{5, 17, 25, 45, 55, 60, 70\}$																			
$\mathcal{U}$	0	1	5	0	1	2	0	3	5	6	7	9	11	12	15	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	4	5	6	7	7

b) The result of divide$(8, 3)$ on the structure of a). Representative value 30 was removed from $\mathcal{R}$. We shifted and updated $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{C}$ to remove the old representative and accommodate for a new element with value 2.

Index containing the sum of the merged indices.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z	5	1	4	7	1	1	6	3	2	1	1	4	1	3	5	10	5	10	2
Y	5	6	10	17	18	19	25	28	30	31	32	36	37	40	45	55	60	70	72
$\mathcal{R}$	$\{5, 17, 25, 45, 55, 60, 70\}$																		
$\mathcal{U}$	0	1	5	0	1	2	0	3	5	6	7	11	12	15	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	3	3	3	3	3	3	3	4	5	6	7	7

c) The result of merge(12) on the structure of c).

Figure 2.1: Illustrating operations on the data structure with $B2^{\delta} = 4$. a) shows the data structure immediately after a rebuild, b) shows the result of performing divide$(8, 3)$ on the structure of a), and c) shows the result of performing merge(12) on the structure of b).

Lemma 2 to store $\mathcal{I}$ is $O(B)$. Instead, to get our desired constant time bounds, we represent $\mathcal{I}$ using a resizable data structure with the same number of elements as Y that supports this kind of update. We must support $\mathsf{select}_{\mathcal{I}}(i)$, $\mathsf{rank}_{\mathcal{I}}(q)$, and $\mathsf{pred}_{\mathcal{I}}(q)$ as well as inserting and deleting elements in constant time. Because $\mathcal{I}$ has few and small elements, we can support the operations in constant time by representing it using a bitstring $\mathcal{B}$ and a structure $\mathcal{C}$ which is the prefix sum over $\mathcal{B}$ as follows.

Let $\mathcal{B}$ be a bitstring of length $|Y| \leq B$, where $\mathcal{B}[i] = 1$ iff there is a representative at index i. $\mathcal{C}$ has $|Y|$ elements, where $\mathcal{C}[i]$ is the prefix sum of $\mathcal{B}$ including element i. Since $\mathcal{C}$ requires $O(B \log B)$ bits in total we can pack it in a single word. We answer queries as follows: $\mathsf{rank}_{\mathcal{I}}(q)$ equals $\mathcal{C}[q-1]$, we answer $\mathsf{select}_{\mathcal{I}}(i)$ by subtracting i from all elements in $\mathcal{C}$ and return one plus the number of elements smaller than 0 (as done in $\mathcal{U}$ when answering search), and we find $\mathsf{pred}_{\mathcal{I}}(q)$ as the index of the least significant bit in $\mathcal{B}$ after having masked all indices larger than q. Updates are performed as follows. Using mask, shift and concatenate operations, we can ensure that $\mathcal{B}$ and $\mathcal{C}$ have the same size as Y at all times (we extend and shrink them when performing divide and merge operations). Inserting or deleting a representative is to set a bit in $\mathcal{B}$, and to keep $\mathcal{C}$ up to date, we employ the same ± 1 update operation as used in $\mathcal{U}$.

We finally need to adjust the relative offsets of all elements with a changed representative in $\mathcal{U}$ (since they now belong to a representative with a different value). In particular, if the representative for $\mathcal{U}[j]$ changed value from v to v', we must subtract $v' - v$ from $\mathcal{U}[j]$. This can be done for all affected elements belonging to a single representative simultaneously in $\mathcal{U}$ by a single addition with an appropriate bitmask (update a range of $\mathcal{U}$). Note that we know the range of elements to update from the representative indices. Finally, we may need to insert or delete an element in $\mathcal{U}$, which can be done easily by mask, shift and concatenate operations on the word $\mathcal{U}$. This leads to Theorem 6.

Theorem 6. *There is a linear space data structure for dynamic partial sums supporting each operation* search*,* sum*,* update*,* insert*,* delete*,* divide*, and* merge *on a sequence of length* $O(\min\{w/\log w, w/\delta\})$ *in worst-case constant time.*

2.3.2 Dynamic Partial Sums for Large Sequences

Willard [Wil00] (and implicitly Dietz [Die89]) showed that a leaf-oriented B-tree with out-degree B of height h can be maintained in $O(h)$ worst-case time if: 1) searches, insertions and deletions take $O(1)$ time per node when no splits or merges occur, and 2) merging or splitting a node of size B requires $O(B)$ time.

We use this as follows, where Z is our integer sequence of length s. Create a leaf-oriented B-tree of degree $B = \Theta(\min\{w/\log w, w/\delta\})$ storing Z in the leaves, with height $h = O(\log_B n) = O(\log n/\log(w/\delta))$. Each node v uses Theorem 6 to store the $O(B)$ sums of leaves in each of the subtrees of its children. Searching for t in a node corresponds to finding the successor $Y[i]$ of t among these sums. Dividing or merging elements in Z corresponds to inserting or deleting a leaf. This concludes the proof of Theorem 2.

2.4 Substring Concatenation

In this section we prove Theorem 3. Recall that we must store a string R subject to substring concatenation queries: given two strings x and y return the location of an occurrence of xy in R or `NO` if no such occurrence exist.

To prove (i) we need the following definitions. For a substring x of R, let $S(x)$ denote the suffixes of R that have x as a prefix, and let $S'(x) = \{i + |x| \mid i \in S(x) \wedge i + |x| \leq n\}$, i.e., $S'(x)$ are the suffixes of R that are immediately preceded by x. Hence for two substrings x and y, the suffixes that have xy as a prefix are exactly $S'(x) \cap S(y)$. We can reduce this intersection problem to a 1D range emptiness problem as follows.

Let $\text{rank}(i)$ be the position of suffix $R[i..r]$ in the lexicographic ordering of all suffixes of R, and let $\text{rank}(A) = \{\text{rank}(i) \mid i \in A\}$ for $A \subseteq \{1, 2, \ldots, n\}$. Then xy is a substring of R if and only if $\text{rank}(S'(x)) \cap \text{rank}(S(y)) \neq \emptyset$. Note that $\text{rank}(S(y))$ is a range $[a, b] \subseteq [1, n]$, and we can determine this range in constant time for any substring y using a constant-time weighted ancestor query on the suffix tree of R [GLN14]. Consequently, we can decide if xy is a substring of R by a 1D range emptiness query on the set $\text{rank}(S'(x))$.

Belazzougui et al. [BBPV10] (see also [GGLP15]) recently gave a 1D range emptiness data structure for a set $A \subseteq [1, r]$ using $O(|A| \log^\epsilon r)$ *bits* of space, for any constant $\epsilon > 0$, and answering queries in constant time. We will build this data structure for $\text{rank}(S'(x))$, but doing so for all substrings would require space $\tilde{\Omega}(r^2)$.

To arrive at the space bound of $O(r \log^\epsilon r)$ (words), we employ a heavy path decomposition [HT84] on the suffix tree of R, and only build the data structure for substrings of R that correspond to the top of a heavy path. In this way, each suffix will appear in at most $\log r$ such data structures, leading to the claimed $O(r \log^\epsilon r)$ space bound (in words). In addition, we build a $O(r)$-space nearest common ancestor data structure [HT84] for the suffix tree of R. Constant-time nearest common ancestor queries will allow us to also answer longest common prefix queries on R in constant time.

To answer a substring concatenation query with substrings x and y, we first determine how far y follows the heavy path in the suffix tree from the location where x stops. This can be done in $O(1)$ time by a constant-time longest common prefix query between two suffixes of R. We then proceeed to the top of the next heavy path, where we query the 1D range reporting data structure with the range $\text{rank}(S(y'))$ where y' is the remaining unmatched suffix of y. This completes the query, and the proof of (i).

The second solution (ii) is an implication of a result by Bille et al. [BGVV14]. Given the suffix tree ST_R of R, an *unrooted longest common prefix query* [CGL04] takes a suffix y and a location ℓ in ST_R (either a node or a position on an edge) and returns the location in ST_S that is reached after matching y starting from location ℓ. A substring concatenation query is straightforward to implement using two unrooted longest common prefix queries, the first one starting at the root, and the second starting from the location returned by the first query. It follows from Bille et al. [BGVV14] that we can build a linear space data structure that supports unrooted longest common prefix queries in time $O(\log \log r)$ thus completing the proof of (ii).

2.5 Extensions

In this section we show how to solve two other variants of the dynamic relative compression problem. We first prove Theorem 4, showing how to improve the query time if only supporting operations access and replace. We then show Theorem 5, generalising the problem to support multiple strings. These data structures use the same substring concatenation data structure of Theorem 3 as before but replaces the dynamic partial sums data structure.

2.5.1 Dynamic Relative Compression with Access and Replace

In this setting we constrain the operations on S to $\mathsf{access}(i)$ and $\mathsf{replace}(i, \alpha)$. Then, instead of maintaining a dynamic partial sums data structure over the lengths of the substrings in C, we only need a dynamic predecessor data structure over the prefix sums. The operations are implemented as before, except that for $\mathsf{access}(i)$ we obtain block b_j by computing the predecessor of i in the predecessor data structure, which also immediately gives us access to the local index in b_j. For $\mathsf{replace}(i, \alpha)$, a constant number of updates to the predecessor data structure is needed to reflect the changes. We use substring concatenation queries to restore maximality as described in Section 2.2. The prefix sums of the subsequent blocks in C are preserved since $|b_j| = |b_j^1| + |b_j^2| + |b_j^3|$.

With a linear space implementation of the van Emde Boas data structure [vEB77, vEBKZ77, MN90] we can support the predecessor queries and updates in $O(\log \log N)$ expected time. For substring concatenation we apply Theorem 3(ii) using $O(r)$ space and $O(\log \log r)$. Since the length of source string does not change, we can always assume that $r > N$, and the total time becomes $O(\log \log N + \log \log r) = O(\log \log N)$. In summary, this proves Theorem 4.

2.5.2 Dynamic Relative Compression with Split and Concatenate

Consider the variant of the dynamic relative compression problem where we want to maintain a relative compression of a set of strings $S_1, \ldots, S_k$. Each string S_i has a cover C_i and all strings are compressed relative to the same string R. In this setting $n = \sum_{i=1}^{k} |C_i|$. In addition to the operations access, $\mathsf{replace}$, insert, and delete, we also want to support split and concatenation of strings. Note that the semantics of the operations change to indicate the string(s) to perform a given operation on.

We build a leaf-oriented height-balanced binary tree T_i (e.g. an AVL tree or red-black tree) over the blocks $C_i[1], \ldots, C_i[|C_i|]$ for each string S_i. In each internal node v, we store the sum of the block sizes represented by its leaves. Since the total number of blocks is n, the trees use $O(n)$ space. All operations rely on the standard procedures for searching, inserting, deleting, splitting and joining height-balanced binary trees. All of these run in $O(\log n)$ time for a tree of size n. See for example [CLRS01] for details on how red-black trees achieve this.

The answer to an $\mathsf{access}(i, j)$ query is found by doing a top-down search in T_i using the sums of block sizes to navigate. Since the tree is balanced and the size of the cover is at most n, this takes $O(\log n)$ time. The operations $\mathsf{replace}(i, j, \alpha)$, $\mathsf{insert}(i, j, \alpha)$, and $\mathsf{delete}(i, j)$ all initially require that we use $\mathsf{access}(i, j)$ to locate the block containing the j-th character of S_i. To reflect possible changes to the blocks of the cover, we need to modify the corresponding tree to contain more leaves and restore the balancing property. Since the number of nodes added to the tree is constant these operations each take $O(\log n)$ time. The $\mathsf{concat}(i, j)$ operation requires that we join two trees in the standard way and restore the balancing property of the resulting tree. For the $\mathsf{split}(i, j)$ operation we first split the block that contains position j such that the j-th character is the trailing character of a block. We then split the tree into two trees separated by the new block. This takes $O(\log n)$ time for a height-balanced tree.

To finalize the implementation of the operations, we must restore the maximality property of the affected covers as described in Section 2.2. At most a constant number of blocks are non-maximal as a result of any of the operations. If two blocks can be combined to one, we delete the leaf that represents the rightmost block, update the leftmost block to reflect the change, and restore the property that the tree is balanced. If the tree subsequently contains an internal node with only one

child, we delete it and restore the balancing. Again, this takes $O(\log n)$ time for balanced trees, which concludes the proof of Theorem 5.

2.6 Conclusion

We have shown how to compress a text relatively to a reference string while supporting access to the text and a range of dynamic operations under some strong guarantees for the space usage and the query times. There are, however, room for improvement.

Our solution to DRC is built on data structures for the partial sums problem and the substring concatenation problem. Our partial sums-solution is optimal, but in order to get the desired constant query time for substring concatenation, our data structure uses $O(r \log^{\epsilon} r)$ space. As opposed to this, our linear space solution leads to $O(\log \log r)$ query time. We leave as an open problem if it is possible to get $O(1)$ time substring concatenation queries using $O(r)$ space, which will also carry over to a stronger result for the DRC problem.

Moreover, the size of the cover that is maintained by our DRC data structure is also an interesting parameter. Currently we maintain a 2-approximation of the optimal cover. It would be useful to know if a better approximation ratio can be maintained under the same (or better) time and space bounds that we give.

Acknowledgments We thank Pawel Gawrychowski for helpful discussions.

Chapter 3

Subsequence Automata with Default Transitions

Subsequence Automata with Default Transitions

Philip Bille Inge Li Gørtz Frederik Rye Skjoldjensen

Technical University of Denmark
{phbi,inge,fskj}@dtu.dk

Abstract

Let S be a string of length n with characters from an alphabet of size σ. The *subsequence automaton* of S (often called the *directed acyclic subsequence graph*) is the minimal deterministic finite automaton accepting all subsequences of S. A straightforward construction shows that the size (number of states and transitions) of the subsequence automaton is $O(n\sigma)$ and that this bound is asymptotically optimal.

In this paper, we consider subsequence automata with *default transitions*, that is, special transitions to be taken only if none of the regular transitions match the current character, and which do not consume the current character. We show that with default transitions, much smaller subsequence automata are possible, and provide a full trade-off between the size of the automaton and the *delay*, i.e., the maximum number of consecutive default transitions followed before consuming a character.

Specifically, given any integer parameter k, $1 < k \leq \sigma$, we present a subsequence automaton with default transitions of size $O(nk \log_k \sigma)$ and delay $O(\log_k \sigma)$. Hence, with $k = 2$ we obtain an automaton of size $O(n \log \sigma)$ and delay $O(\log \sigma)$. At the other extreme, with $k = \sigma$, we obtain an automaton of size $O(n\sigma)$ and delay $O(1)$, thus matching the bound for the standard subsequence automaton construction. Finally, we generalize the result to multiple strings. The key component of our result is a novel hierarchical automata construction of independent interest.

3.1 Introduction

Let S be a string of length n with characters from an alphabet of size σ. A *subsequence* of S is any string obtained by deleting zero or more characters from S. The *subsequence automaton* (often called the *directed acyclic subsequence graph*) is the minimal deterministic finite automaton accepting all subsequences of S. Baeza-Yates [BY91] initiated the study of subsequence automata. They presented a simple construction using $O(n\sigma)$ size (size denotes the total number of states *and* transitions) and showed that this bound is optimal in the sense that there are subsequence automata of size at least $\Omega(n\sigma)$. They also considered variations with encoded input strings and multiple strings. Subsequently, several researchers have further studied subsequence automata (and its variants) [TS05, CMT03, CT99, Tro01a, HSTA00, FFMR10, BIST03, Tro99]. See also the surveys by Troníček [Tro01b, Tro03]. The general problem of *subsequence indexing*, not limited to automata based solutions, is investigated by Bille et al. [BFC08].

In this paper, we consider subsequence automata in the context of *default transitions*, that is, special transitions to be taken only if none of the regular transitions match the current character, and which do not consume the current character. Each state has at most one default transition

and hence the automaton remains deterministic. The key point of default transitions is to reduce the size of standard automata at the cost of introducing a *delay*, i.e., the maximum number of consecutive default transition followed before consuming a character. For instance, given a pattern string of length m the classic Knuth-Morris-Pratt (KMP) [KJHMP77] string matching algorithm may be viewed as an automaton with default transitions (typically referred to as *failure transitions*). This automaton has size $O(m)$, whereas the standard automaton with no default transitions would need $\Theta(m\sigma)$ space. The delay of the automaton in the KMP algorithm is either $O(m)$ or $O(\log m)$ depending on the version. Similarly, the Aho-Corasick string matching algorithm for multiple strings may also be viewed as an automaton with default transitions [AC75]. More recently, default transitions have also been used extensively to significantly reduce sizes of deterministic automata for regular expression [KDY$^+$06, HL07]. The main idea is to effectively enable states with large overlapping identical sets of outgoing transitions to "share" outgoing transitions using default transitions.

Surprisingly, no non-trivial bounds for subsequence automata with default transitions are known. Naively, we can immediately obtain an $O(n\sigma)$ size solution with $O(1)$ delay by using the standard subsequence automaton (without default transitions). At the other extreme, we can build an automaton with $n+1$ states (each corresponding to a prefix of S) with a standard and a default transition from the state corresponding to the ith prefix to the state corresponding to the $i+1$st prefix (the standard transition is labeled $S[i+1]$). It is straightforward to show that this leads to an $O(n)$ size solution with $O(n)$ delay. Our main result is a substantially improved trade-off between the size and delay of the subsequence automaton:

Theorem 7. *Let S be a string of n characters from an alphabet of size σ. For any integer parameter k, $1 < k \leq \sigma$, we can construct a subsequence automaton with default transitions of size $O(nk \log_k \sigma)$ and delay $O(\log_k \sigma)$.*

Hence, with $k = 2$ we obtain an automaton of size $O(n \log \sigma)$ and delay $O(\log \sigma)$. At the other extreme, with $k = \sigma$, we obtain an automaton of size $O(n\sigma)$ and delay $O(1)$, thus matching the bound for the standard subsequence automaton construction.

To obtain our result, we first introduce the *level automaton*. Intuitively, this automaton uses the same states as the standard solution, but hierarchically orders them in a tree-like structure and samples a selection of their original transitions based on their position in the tree, and adds a default transition to the next state on a higher level. We show how to do this efficiently leading to a solution with $O(n \log n)$ size and $O(\log n)$ delay. To achieve our full trade-off from Theorem 1 we show how to augment the construction with additional ideas for small alphabets and generalize the level automaton with parameter k, $1 < k \leq \sigma$, where large k reduces the height of the tree but increases the number of transitions. In the final section we generalize the result to *multiple* strings.

3.2 Preliminaries

A *deterministic finite automaton* (DFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a set of nodes called *states*, δ is a set of labeled directed edges between states, called *transitions*, where each label is a character from the alphabet Σ, $q_0 \in Q$ is the *initial* state and $F \subseteq Q$ is a set of *accepting* states. No two outgoing transitions from the same state have the same label. The DFA is *incomplete* in the sense that every state does not contain transitions for every character in Σ. The *size* of A is the sum of the number of states and transitions.

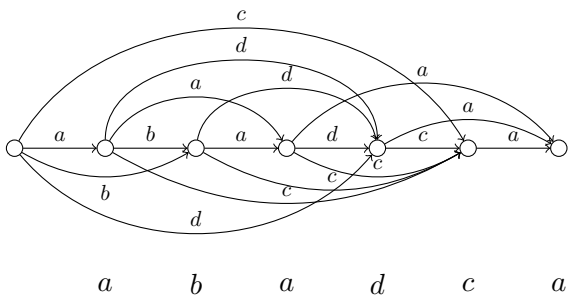

Figure 3.1: An example of an SA constructed from the string *abadca*.

We can think of A as an *edge-labeled directed graph*. Given a string P and a path p in A we say that p and P match if the concatenation of the labels on the transitions in p is P. We say that A *accepts* a string P if there is a path in A, from q_0 to any state in F, that matches P. Otherwise A *rejects* P.

A *deterministic finite automaton with default transitions* is a deterministic finite automaton AD where each state can have a single unlabeled default transition. Given a string P and a path p in AD we define a match between P and p as before, with the exception that for any default transition t in p the corresponding character in P cannot match any standard transition out of the source state of t. Definition of accepted and rejected strings are as before. The *delay* of AD is the maximum length of any path matching a single character, i.e., if the delay of AD is d then we follow at most $d-1$ default transitions for every character that is matched in P.

A *subsequence* of S is a string P, obtained by removing zero or more occurrences of characters from S. The alphabet of the string S is denoted by $\Sigma(S)$. A *subsequence automaton* constructed from S, is a deterministic finite automaton that accepts string P iff P is a subsequence of S. A subsequence automaton construction is presented in [BY91]. This construction is often called the *directed acyclic subsequence graph* or DASG, but here we denote it SA. The SA has $n+1$ states, all accepting, that we identify with the integers $\{0, 1, \ldots, n\}$. For each state s, $0 \leq s \leq n$, we have the following transitions:

- For each character α in $\Sigma(S[s+1, n])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.

The SA has size $O(n\sigma)$ since every state can have at most σ transitions. An example of an SA is given in Figure 3.1.

A *subsequence automaton with default transitions* constructed from S, denoted SAD, is a deterministic finite automaton with default transitions that accepts string P iff P is a subsequence of S.

The next section explores different configurations of transitions and default transitions in SADs.

3.3 New Trade-Offs for Subsequence Automata.

We now present a new trade-off for subsequence automata, with default transitions. We will gradually refine our construction until we obtain an automaton that gives the result presented in Theorem 1. In each construction we have $n+1$ states that we identify with the integers $\{0, 1, \ldots, n\}$.

Each of these states represents a prefix of the string S and are all accepting states. We first present the *level automaton* that gives the first non-trivial trade-off that exploits default transitions. The general idea is to construct a hierarchy of states, such that every path that only uses default transitions is guaranteed to go through states where the outdegree increases at least exponentially. The level automaton is a SAD of size $O(n \log n)$ and delay $O(\log n)$. By arguing that any path going through a state with outdegree σ will do so by taking a regular transition, we are able to improve both the size and delay of the level automaton. This results in the *alphabet-aware level automaton* which is a SAD of size $O(n \log \sigma)$ and delay $O(\log \sigma)$. Finally we present a generalized construction that gives a trade-off between size and delay by letting parameter k, $1 < k \leq \sigma$, be the base of the exponential increase in outdegree on paths with only default transitions. This SAD has size $O(nk \log_k \sigma)$ and delay $O(\log_k \sigma)$. With $k = 2$ we get an automaton of size $O(n \log \sigma)$ and delay $O(\log \sigma)$. At the other extreme, for $k = \sigma$ we get an automaton of size $O(n\sigma)$ and delay $O(1)$.

3.3.1 Level Automaton

The level automaton is a SAD with $n+1$ states that we identify with the integers $\{0, 1, \ldots, n\}$. All states are accepting. For each state $i > 0$, we associate a level, level(i), given by:

$$\text{level}(i) = \max(\{x \mid i \bmod 2^x = 0\})$$

Hence, level(i) is the exponent of the largest power of two that divides i. The level function is in the literature known as the ruler function. We do not associate any level with state 0. Note that the maximum level of any state is $\log_2 n$. For a nonnegative integer s, we define $\overline{s}$ to be the smallest integer $\overline{s} > s$ such that level($\overline{s}$) $\geq$ level(s) $+ 1$.

The transitions in the level automaton are as follows: From state 0 we have a default transition to state 1 and a regular transition to state 1 with label $S[1]$. For every other state s, $1 \leq s \leq n$, we have the following transitions:

- A default transition to state $\overline{s}$. If no such state exist, the state s does not have a default transition.
- For each character α in $\Sigma(S[s+1, \min(\overline{s}, n)])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.

An example of the level automaton constructed from the string *abacbabcabad* and alphabet $\{a, b, c, d\}$ is given in Figure 3.2. The dashed arrows denote default transitions and the vertical position of the states denotes their level.

We first show that the level automaton is a SAD for S, i.e., the level automaton accepts a string iff the string is a subsequence of S. To do so suppose that P is a string of length m accepted by the level automaton and let $s_1, s_2, \ldots, s_m$ be the sequence of states visited with regular transitions on the path that accepts P. From the definition of the transition function, we know that if a transition with label α leads to state s', then $S[s'] = \alpha$. This means that $S[s_1]S[s_2] \ldots S[s_m]$ spells out a subsequence of S if the sequence $s_1, s_2, \ldots, s_m$ is strictly increasing. From the definition of the transitions, a state s only have transitions to states s' if $s' > s$. Hence, the sequence is strictly increasing.

For the other direction, we show that the level automaton simulates the SA. At each state s, trying to match character α, we find the smallest state $s' > s$ such that s' has an incoming transition

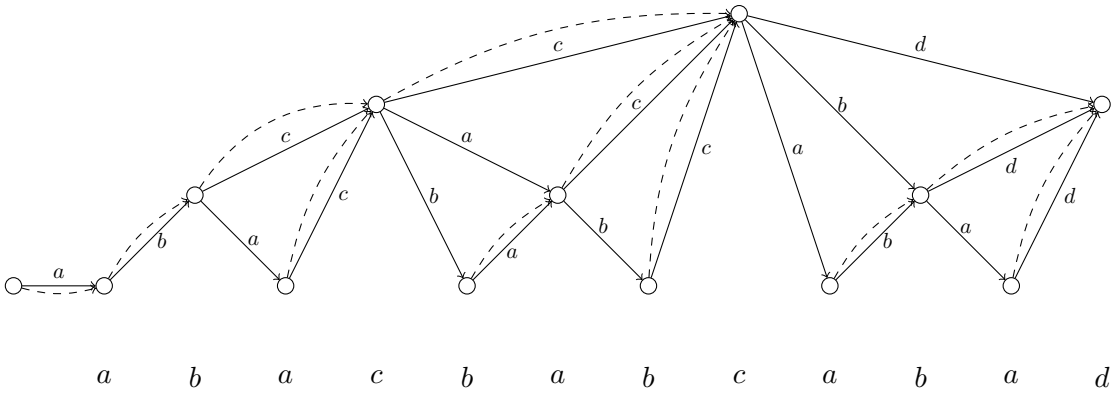

Figure 3.2: The level automaton constructed from the string *abacbabcabad*.

with label α: By the construction, either s has an outgoing transition leading directly to s' *or* we follow default transitions until reaching the first state with a transition to s'. This means that the states visited with standard transitions in the level automaton are the same states that are visited in the SA. Since the SA accepts all subsequences of S this must also hold for the level automaton.

3.3.1.1 Analysis

The following shows that the number of outgoing transitions increase with a factor two when the level increase by one. For all $s > 0$, we have the following property of $\overline{s}$ and level(s):

$$\overline{s} - s = 2^{\text{level}(s)} \tag{3.1}$$

By definition, $2^{\text{level}(s)}$ divides s. This means that we can write s as $c \cdot 2^{\text{level}(s)}$, where c is a uneven positive integer. We know that c is uneven because $2^{\text{level}(s)}$ is the largest power of two that divides s. The next integer, larger than s, that $2^{\text{level}(s)}$ divides is $s' = s + 2^{\text{level}(s)}$. This means that $\overline{s} \geq s'$. We can rewrite s' as follows: $s' = s + 2^{\text{level}(s)} = c \cdot 2^{\text{level}(s)} + 2^{\text{level}(s)} = (c+1) \cdot 2^{\text{level}(s)}$. Since c is uneven we know that $c + 1$ is even so we can rewrite s' further: $s' = \frac{(c+1)}{2} \cdot 2^{\text{level}(s)+1}$. This shows that $2^{\text{level}(s)+1}$ divides s' which means that $s' = \overline{s}$ and we conclude that $\overline{s} - s = 2^{\text{level}(s)}$.

Since the maximal level of any state is $\log_2 n$ and the level increase every time we follow a default transition, the delay of the level automaton is $O(\log n)$.

At each level l we have $O(n/2^{l+1})$ states, since every 2^lth state is divided by 2^l, and 2^l is the largest divisor in every second of these cases. Since $\overline{s} - s = 2^{\text{level}(s)}$ each state at level l has at most 2^l+1 outgoing transitions. Therefore, each level contribute with size at most $n/2^{l+1} \cdot (2^l+1) = O(n)$. Since we have at most $O(\log n)$ levels, the total size becomes $O(n \log n)$.

In summary, we have shown the following result.

Lemma 3. *Let S be a string of n characters. We can construct a subsequence automaton with default transitions of size $O(n \log n)$ and delay $O(\log n)$.*

3.3.2 Alphabet-aware level automaton

We introduce the *Alphabet-aware level automaton*. When the level automaton reaches a state s where $\overline{s} - s \geq \sigma$, then s can have up to σ outgoing transitions without violating the space analysis

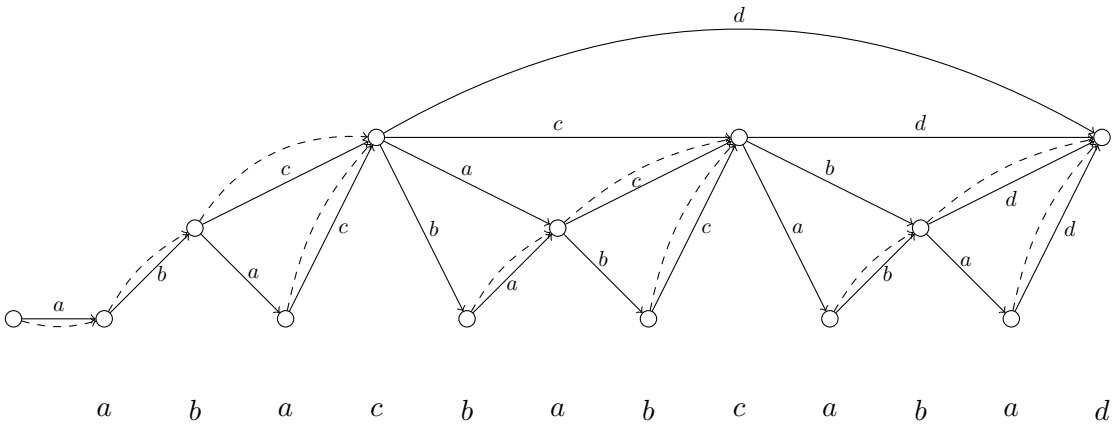

Figure 3.3: The alphabet-aware level automaton constructed from the string *abacbabcabad*.

from the previous section. The level automaton only has a transition for each character in $\Sigma(S[s+1, \min(\overline{s}, n)])$. Hence, for all states s in the alphabet-aware level automaton where $\overline{s} - s \geq \sigma$, we let s have a transition for each symbol α in Σ, to the smallest state $s' > s$ such that $S[s'] = \alpha$. No matching path can take a default transition from a state with σ outgoing transitions. Hence, states with σ outgoing transitions do not need default transitions.

We change the level function to reflect this. For each state $1 \leq i \leq n$ we have that:

$$\text{level}(i) = \min(\lceil \log_2 \sigma \rceil, \max(\{x \mid i \bmod 2^x = 0\}))$$

The transitions in the alphabet-aware level automaton is as follows: From state 0 we have a default transition to state 1 and a regular transition to state 1 with label $S[1]$. For every other state s, $1 \leq s \leq n$, we have the following transitions:

- A default transition to state $\overline{s}$. If no such state exist, the state s does not have a default transition.
- If $\overline{s} - s < \sigma$ then for each character α in $\Sigma(S[s+1, \min(\overline{s}, n)])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.
- If $\overline{s} - s \geq \sigma$ then for each character α in $\Sigma(S[s+1, n])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.

An example of the alphabet-aware level automaton constructed from the string *abacbabcabad* and alphabet $\{a, b, c, d\}$ is given in Figure 3.3. The level automaton in Figure 3.2 is constructed from the same string and the same alphabet. For comparison, in Figure 3.3 state 4 now has outdegree σ and has transitions to the first succeeding occurrence of any unique character and state 8 has been constrained to level $\lceil \log_2 \sigma \rceil$.

The alphabet-aware level automaton is a SAD by the same arguments that led to Lemma 3.

The delay is now bounded by $O(\log \sigma)$ since no state is assigned a level higher than $\lceil \log_2 \sigma \rceil$. The size of each level is still $O(n)$. Hence, the total size becomes $O(n \log \sigma)$

In summary, we have shown the following result.

Lemma 4. *Let S be a string of n characters. We can construct a SAD of S with size $O(n \log \sigma)$ and delay $O(\log \sigma)$.*

3.3.3 Full trade-off

We can generalize the construction above by introducing parameter k, $1 < k \leq \sigma$, which is the base of the exponential increase in outdegree of states on every path that only uses default transitions. Now, when we follow a default transition from s to $\overline{s}$, the number of outgoing transitions increase with a factor k instead of a factor 2. This gives a trade-off between size and delay in the SAD determined by k. Increasing k gives a shorter delay of the SAD but increases the size and vice versa.

Each state, except state 0, is still associated with a level, but we need to generalize the level function to account for the parameter k. For every k and i we have that:

$$\text{level}(i,k) = \min(\lceil \log_k \sigma \rceil, \max(\{x \mid i \bmod k^x = 0\}))$$

Now, the level function gives the largest power of k that divides i.

The transitions in the generalized alphabet-aware level automaton is as follows: From state 0 we have a default transition to state 1 and a regular transition to state 1 with label $S[1]$. For every other state s, $1 \leq s \leq n$, we have the following transitions:

- A default transition to state $\overline{s}$. If no such state exist, the state s does not have a default transition.
- If $\overline{s} - s < \sigma$ then for each character α in $\Sigma(S[s+1, \min(\overline{s}, n)])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.
- If $\overline{s} - s \geq \sigma$ then for each character α in $\Sigma(S[s+1, n])$, there is a transition labeled α to the smallest state $s' > s$ such that $S[s'] = \alpha$.

We can show that the generalized alphabet-aware level automaton is a SAD by the same arguments that led to Lemma 4.

3.3.3.1 Analysis

The delay is bounded by $O(\log_k \sigma)$ because no state is assigned a level higher than $\lceil \log_k \sigma \rceil$.

With the new definition of the level function we have that

$$\overline{s} - s \leq k^{\text{level}(s,k)+1}$$

for all $s > 0$. This expression bounds the number of outgoing transitions from state s.

At level l we have $O(n(k-1)/(k^{l+1}))$ states each with $O(k^{l+1})$ outgoing transitions such that each level has size $O(nk)$. The size of the automaton becomes $O(nk \log_k \sigma)$ because we have $O(\log_k \sigma)$ levels.

In summary, we have shown Theorem 1.

3.4 Subsequence automata for multiple strings

Crochemore et al. [CMT03] generalizes the simple subsequence automaton to multiple strings. Given a set of strings $\mathcal{S} = \{S_1, S_2, \ldots S_N\}$ of length $n_1, n_2, \ldots, n_N$, two types of automata are presented: The subsequence automaton accepts a pattern P iff P is a subsequence of *some* string in $\mathcal{S}$ and

the *common* subsequence automaton accepts P iff P is a subsequence of *every* string in $\mathcal{S}$. When $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ and $n_1 = n_2 = \ldots = n_N = n$, a $\Omega(n^N/(N+1)^N N!))$ lower bound on the number of states is shown for the subsequence automaton [TS05]. For both automata, the number of states is trivially upper bounded by $O(n_1 \cdot n_2 \cdot \ldots \cdot n_N)$ such that the total size becomes $O(\sigma \cdot n_1 \cdot n_2 \cdot \ldots \cdot n_N)$. We can reduce the size of these automata by augmenting them with default transitions. The following generalization is in the same spirit as the naive generalization of the single string automaton in section 3.1: We introduce default transitions and save a factor σ in the space but also introduce a delay. Consider the naive common subsequence automaton with default transitions: For two strings S_1, S_2 we have $n_1 \cdot n_2 + 1$ states that we identify with the points $\{1, \ldots, n_1\} \times \{1, \ldots, n_2\} \cup \{(0,0)\}$. For each state (s_1, s_2) we have the following transitions:

- A default transition to state $(s_1 + 1, s_2 + 1)$. If no such state exist, the state (s_1, s_2) does not have a default transition.
- If character $S_1[s_1 + 1]$ is in $\Sigma(S_2[s_2 + 1, n_2])$, there is a transition labeled $S_1[s_1 + 1]$ to the state $(s_1 + 1, s_2')$ such that $s_2' > s_2$ is the minimal index where $S_2[s_2'] = S_1[s_1 + 1]$.
- If character $S_2[s_2 + 1]$ is in $\Sigma(S_1[s_1 + 1, n_1])$, there is a transition labeled $S_2[s_2 + 1]$ to the state $(s_1', s_2 + 1)$ such that $s_1' > s_1$ is the minimal index where $S_1[s_1'] = S_2[s_2 + 1]$.

The states of the automaton represents the progression in S_1 and S_2, such that state (s_1, s_2) represents that subsequences of the prefixes $S_1[1, s_1]$ and $S_2[1, s_2]$ have been used to match a prefix of P. Each state (s_1, s_2) considers the symbols $S_1[s_1 + 1]$ and $S_2[s_2 + 1]$ for matching with the next symbol in P. If this is not possible, a default transition is followed to state $(s_1 + 1, s_2 + 1)$.

For this automaton the size is $O(n_1 \cdot n_2)$ and the delay is $O(\min(n_1, n_2))$. Hence, we save a factor σ in the size, but introduce a significant delay. As we did for the subsequence automaton for a single string, we introduce a level automaton that associates a level with each state. In this way we are able to reduce the delay significantly with only a small increase in size. For simplicity we only present our construction for the common subsequence automaton, but it follows immediately that the a similar construction is also applicable to the subsequence automaton.

3.4.1 The alphabet-aware level automaton for two strings

The alphabet-aware level automaton for two strings S_1, S_2 have $n_1 \cdot n_2 + 1$ states that we identify with the points $\{1, \ldots, n_1\} \times \{1, \ldots, n_2\} \cup \{(0,0)\}$. We define the *diagonal* of a state (i, j), as the set of states $\{(i + k, j + k) \mid 0 < i + k \leq n_1 \text{ and } 0 < j + k \leq n_2\}$. We say that states belong to the same diagonal if the diagonals of the states defines identical sets of states. For states $(s_1, s_2), (s_1', s_2')$ in the same diagonal we define an ordering: $(s_1, s_2) < (s_1', s_2')$ if $s_1 < s_1'$. This ordering defines a unique position for each state in its diagonal. For each state (s_1, s_2) we associates the integer $\min(s_1, s_2)$, such that $(s_1, s_2) - (s_1', s_2') = \min(s_1, s_2) - \min(s_1', s_2')$ and $(s_1, s_2) \bmod x = \min(s_1, s_2) \bmod x$. With each diagonal of states, we associate a level structure identical to the one used in the alphabet-aware level automaton for a single string. For each state (i, j), $1 \leq i \leq n_1, 1 \leq j \leq n_2$, we again associate a level:

$$\text{level}((i,j)) = \min(\lceil \log_2 \sigma \rceil, \max(\{x \mid (i,j) \bmod 2^x = 0\}))$$

For a state (i, j), we define $\overline{(i,j)}$ to be the smallest state in the diagonal of (i, j) such that $\overline{(i,j)} > (i, j)$ and $\text{level}(\overline{(i,j)}) > \text{level}((i,j))$. The state $\overline{(i,j)}$ is not guaranteed to exist.

The alphabet-aware level automaton for two strings S_1, S_2 has the following transitions: State $(0,0)$ has a transition labeled $S_1[1]$ to state $(1, s_2)$ where s_2 is the minimal index such that $S_2[s_2] = S_1[1]$, a transition labeled $S_2[1]$ to state $(s_1, 1)$ where s_1 is the minimal index such that $S_1[s_1] = S_2[1]$ and a default transition to state $(1,1)$. The regular transitions only exists if the states $(1, s_2)$ and $(s_1, 1)$ exists. Every other state (s_1, s_2), where $(\overline{s_1}, \overline{s_2}) = \overline{(s_1, s_2)}$, have the following transitions:

- A default transition to state $\overline{(s_1, s_2)}$. If no such state exist, the state (s_1, s_2) does not have a default transition.
- If $\overline{(s_1, s_2)} - (s_1, s_2) < \sigma$ then for each character α in $(\Sigma(S_1[s_1 + 1, \min(\overline{s_1}, n_1)]) \cup \Sigma(S_2[s_2 + 1, \min(\overline{s_2}, n_2)])) \cap \Sigma(S_1[s_1 + 1, n_1]) \cap \Sigma(S_2[s_2 + 1, n_2])$, there is a transition labeled α to the state (s_1', s_2'), where $s_1' > s_1$ and $s_2' > s_2$ are the minimal indices such that $S_1[s_1'] = S_2[s_2'] = \alpha$.
- If $\overline{(s_1, s_2)} - (s_1, s_2) \geq \sigma$ then for each character α in $\Sigma(S_1[s_1 + 1, n_1]) \cap \Sigma(S_2[s_2 + 1, n_2])$, there is a transition labeled α to the state (s_1', s_2'), where $s_1' > s_1$ and $s_2' > s_2$ are the minimal indices such that $S_1[s_1'] = S_2[s_2'] = \alpha$.

The intuition behind the expression $(\Sigma(S_1[s_1 + 1, \min(\overline{s_1}, n_1)]) \cup \Sigma(S_2[s_2 + 1, \min(\overline{s_2}, n_2)])) \cap \Sigma(S_1[s_1 + 1, n_1]) \cap \Sigma(S_2[s_2 + 1, n_2])$ is as follows: The union gives the set of unique characters that exists in the interval of the strings that the default transition will skip. But only the characters that exists in the suffix of both strings needs a transition.

An example of an incomplete common subsequence automaton for two strings is given in Figure 3.4.

3.4.1.1 Analysis

For a state (i, j), if $\overline{(i, j)}$ exists, we have the following property:

$$\overline{(i, j)} - (i, j) = 2^{\text{level}((i,j))}$$

This property follows from the same argument that led to equation (3.1). The number of transitions out of every state s, is now bounded by $2 \cdot 2^{\text{level}(s)}$ since both S_1 and S_2 can contribute with up to $2^{\text{level}(s)}$ transitions.

We can calculate the size of the alphabet-aware level automaton for two strings by summing up the space contribution from each diagonal of states. Let d be a diagonal consisting of $|d|$ states. Then the size of d is $O(|d| \log \sigma)$, since each diagonal has the size of an alphabet-aware level automaton for one string. If D is the set of all diagonals, then the total size of the automaton becomes

$$O\left(\sum_{d \in D} |d| \log \sigma\right) = O\left(\log \sigma \cdot \sum_{d \in D} |d|\right) = O(\log \sigma \cdot n_1 \cdot n_2)$$

The last step exploits that the sum of the states in all diagonals is exactly the number of states in the automaton. In summary we have shown the following result:

Lemma 5. *Let S_1, S_2 be strings of length n_1 and n_2 over an alphabet of size σ. We can construct a subsequence automaton and a common subsequence automaton with default transitions of size $O(n_1 n_2 \log \sigma)$ and delay $O(\log \sigma)$.*

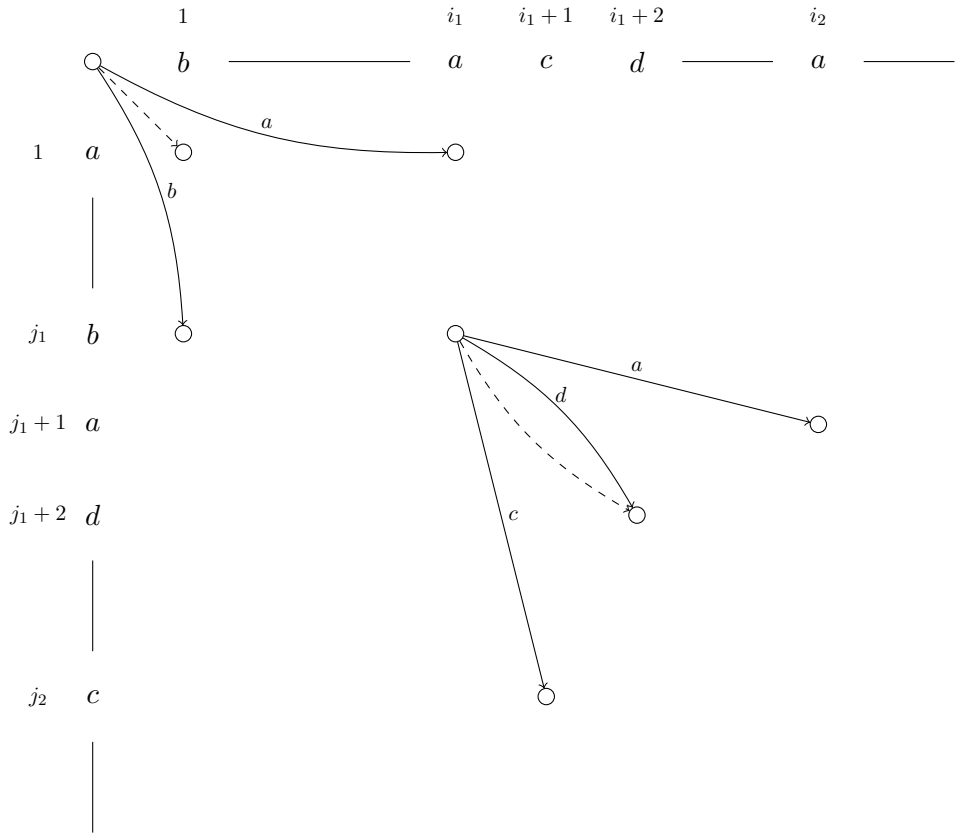

Figure 3.4: An incomplete common subsequence automaton for two strings S_1, S_2, laid out in a two-dimensional grid(S_1 horizontally and S_2 vertically). State $(0, 0)$ has a transition labeled $a = S_2[1]$ to state $(i_1, 1)$ and a transition labeled $b = S_1[1]$ to state $(1, j_1)$. State (i_1, j_1) is at level 1 and has a transition for each unique character in $\Sigma(S_1[i_1 + 1, i_1 + 2]) \cup \Sigma(S_2[j_1 + 1, j_1 + 2]) = \{a, c, d\}$. This figure depicts a situation where $\{a, c, d\} \subseteq (\Sigma(S_1[i_1 + 1, n_1]) \cap \Sigma(S_2[j_1 + 1, n_2]))$. Transitions out of the remaining states are missing from the illustration.

3.4.2 The alphabet-aware level automaton for multiple strings

The alphabet-aware level automaton for the set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$ have $1 + \prod_{i=1}^{N} S_i$ states that we identify with the set of integer points $\{1, \ldots, n_1\} \times \{1, \ldots, n_2\} \times \ldots \times \{1, \ldots, n_N\} \cup \{(0, 0, \ldots, 0)\}$. Hence, a state in the automaton corresponds to a tuple with N elements $(s_1, s_2, \ldots, s_N)$. We generalize the definition of diagonals for dimension N as follows. The diagonal of a state $(s_1, s_2, \ldots, s_N)$ is the set of states:

$$\{(s_1 + k, s_2 + k, \ldots, s_N + k) \mid \bigwedge_{i=1}^{N} 0 < s_i + k \leq n_i\}$$

Again, states belong to the same diagonal if the diagonal of each state defines identical sets of states. For states $(s_1, s_2, \ldots, s_N), (s'_1, s'_2, \ldots, s'_N)$ in the same diagonal we define an ordering: $(s_1, s_2, \ldots, s_N) < (s'_1, s'_2, \ldots, s'_N)$ if $s_1 < s'_1$. This ordering defines a unique position for each state in its diagonal. For each state $(s_1, s_2, \ldots, s_N)$ we associate the integer $\min(s_1, s_2, \ldots, s_N)$ and define subtraction and modulo operations on states, in the same diagonal, as in the previous section.

With each state we again associate the level:

$$\text{level}((s_1, s_2, \ldots, s_N)) = \min(\lceil \log_2 \sigma \rceil, \max(\{x \mid (s_1, s_2, \ldots, s_N) \bmod 2^x = 0\}))$$

For a state $(s_1, s_2, \ldots, s_N)$, we define $\overline{(s_1, s_2, \ldots, s_N)}$ to be the smallest state in the same diagonal such that $\overline{(s_1, s_2, \ldots, s_N)} > (s_1, s_2, \ldots, s_N)$ and $\text{level}(\overline{(s_1, s_2, \ldots, s_N)}) > \text{level}((s_1, s_2, \ldots, s_N))$. The state $\text{level}(\overline{(s_1, s_2, \ldots, s_N)})$ is not guaranteed to exist.

The alphabet-aware level automaton for multiple strings, $S_1, S_2, \ldots, S_N$, has the following transitions: State $(0, 0, \ldots, 0)$ has a transition labeled $S_i[1]$ to state $(s_1, s_2, \ldots, s_N)$ where $s_i = 1$ and s_j is the minimal index where $S_j[s_j] = S_i[1]$, for every $i = \{1, 2, \ldots, N\}$ and $j \neq i$. These transitions only exists if index s_j exists. State $(0, 0, \ldots, 0)$ also have a default transition to state $(1, 1, \ldots, 1)$. Every other state $(s_1, s_2, \ldots, s_N)$, where $(\overline{s_1}, \overline{s_2}, \ldots, \overline{s_N}) = \overline{(s_1, s_2, \ldots, s_N)}$, has the following transitions:

- A default transition to state $\overline{(s_1, s_2, \ldots, s_N)}$. If no such state exist, the state $(s_1, s_2, \ldots, s_N)$ does not have a default transition.
- If $\overline{(s_1, s_2, \ldots, s_N)} - (s_1, s_2, \ldots, s_N) < \sigma$ then for each character α in $(\bigcup_{i=1}^{N} \Sigma(S_i[s_i{+}1, \min(\overline{s_i}, n_i)])) \cap (\bigcap_{i=1}^{N} \Sigma(S_i[s_i{+}1, n_i]))$ there is a transition labeled α to the state $(s'_1, s'_2, \ldots, s'_N)$, where, $s'_i > s_i$ is the minimal index such that $S_i[s'_i] = \alpha$, for all $1 \leq i \leq N$
- If $\overline{(s_1, s_2, \ldots, s_N)} - (s_1, s_2, \ldots, s_N) \geq \sigma$ then for each character α in $\bigcap_{i=1}^{N} \Sigma(S_i[s_i + 1, n_i])$ there is a transition labeled α to the state $(s'_1, s'_2, \ldots, s'_N)$, where $s'_i > s_i$ is the minimal index such that $S_i[s'_i] = \alpha$, for all $1 \leq i \leq N$.

3.4.2.1 Analysis

For each state $(s_1, s_2, \ldots, s_N)$ we have that

$$\overline{(s_1, s_2, \ldots, s_N)} - (s_1, s_2, \ldots, s_N) = 2^{\text{level}((s_1, s_2, \ldots, s_N))}$$

This property follows from the same argument that led to equation (3.1).

The number of transitions out of every state s, is now bounded by $N \cdot 2^{\text{level}(s)}$ because each of the N strings can contribute with up to $2^{\text{level}(s)}$ transitions.

We can calculate the size of the alphabet-aware level automaton for N strings by summing up the space contribution from each diagonal of states. Let d be a diagonal consisting of $|d|$ states. Then the size of d is $O(N|d|\log\sigma)$. If D is the set of all diagonals, then the total size of the automaton becomes

$$O\left(\sum_{d\in D} N|d|\log\sigma\right) = O\left(N\log\sigma\cdot\sum_{d\in D}|d|\right) = O\left(N\log\sigma\cdot\prod_{i=1}^{N} n_i\right)$$

The last step is possible since the sum over the states in all diagonals is the number of states in the automaton. In summary we have shown the following result:

Theorem 8. *Let $S_1, S_2, \ldots S_N$ be a set of strings of length $n_1, n_2, \ldots, n_N$ over an alphabet of size σ. We can construct a subsequence automaton and a common subsequence automaton with default transitions of size $O(N\log\sigma\cdot\prod_{i=1}^{N} n_i)$ and delay $O(\log\sigma)$.*

Chapter 4

Deterministic Indexing for Packed Strings

Deterministic Indexing for Packed Strings

Philip Bille* phbi@dtu.dk
Inge Li Gørtz* inge@dtu.dk
Frederik Rye Skjoldjensen† fskj@dtu.dk

Technical University of Denmark
{phbi,inge,fskj}@dtu.dk

Abstract

Given a string S of length n, the classic string indexing problem is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. In the *deterministic* variant the goal is to solve the string indexing problem without any randomization (at preprocessing time or query time). In the *packed* variant the strings are stored with several character in a single word, giving us the opportunity to read multiple characters simultaneously. Our main result is a new string index in the deterministic *and* packed setting. Given a packed string S of length n over an alphabet σ, we show how to preprocess S in $O(n)$ (deterministic) time and space $O(n)$ such that given a packed pattern string of length m we can support queries in (deterministic) time $O\left(m/\alpha + \log m + \log\log\sigma\right)$, where $\alpha = w/\log\sigma$ is the number of characters packed in a word of size $w = \Theta(\log n)$. Our query time is always at least as good as the previous best known bounds and whenever several characters are packed in a word, i.e., $\log\sigma \ll w$, the query times are faster.

4.1 Introduction

Let S be a string of length n over an alphabet of size σ. The *string indexing problem* is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (returning the number of occurrences of the pattern).

The string indexing problem is a classic well-studied problem in combinatorial pattern matching and the standard textbook solutions are the suffix tree and the suffix array (see e.g., [Gus97, MM93, McC76, Wei73]). A straightforward implementation of suffix trees leads to an $O(n)$ preprocessing time and space solution that given a pattern of length m supports existential and counting queries in time $O(m\log\sigma)$ and reporting queries in time $O(m\log\sigma+\text{occ})$, where occ is the number of occurrences of the pattern. The suffix array implemented with additional arrays storing longest common prefixes leads to a solution that also uses $O(n)$ preprocessing time and space while supporting existential and counting queries in time $O(m+\log n)$ and reporting queries in time $O(m+\log n+\text{occ})$. If

*Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178) and the Advanced Technology Foundation

†Supported by the Danish Research Council (DFF – 1323-00178)

we instead combine suffix trees with perfect hashing [FKS84] we obtain $O(n)$ *expected* preprocessing time and $O(n)$ space, while supporting existential and counting queries in time $O(m)$ and reporting queries in time $O(m+\text{occ})$. The above bounds hold assuming that the alphabet size σ is polynomial in n. If this is not the case, additional time for sorting the alphabet is required [FCFM00]. For simplicity, we adopt this convention in all of the bounds throughout the paper.

In the *deterministic* variant the goal is to solve the string indexing problem without any randomization. In particular, we cannot combine suffix trees with perfect hashing to obtain $O(m)$ or $O(m+\text{occ})$ query times. In this setting Cole et al. [CKL06] showed how to combine suffix trees and suffix array into the *suffix tray* that uses $O(n)$ preprocessing time and space and supports existential and counting queries in $O(m+\log\sigma)$ time and reporting queries in $O(m+\log\sigma+\text{occ})$ time. Recently, the query times were improved by Fischer and Gawrychowski [FG15] to $O(m+\log\log\sigma)$ and $O(m+\log\log\sigma+\text{occ})$, respectively.

In the *packed* variant the strings are given in a *packed representation*, with several characters in a single word [Bil11, BKBB$^+$14, Bel12, TISA16]. For instance, DNA-sequences have an alphabet of size 4 and are therefore typically stored using 2 bits per character with 32 characters in a 64-bit word. On packed strings we can read multiple characters in constant time and hence potentially do better than the immediate $\Omega(m)$ or $\Omega(m+\text{occ})$ lower bound for existential/counting queries and reporting queries, respectively. In this setting Takagi et al. [TISA16] recently introduced the *packed compact trie* that stores packed strings succinctly and also supports dynamic insertion and deletions of strings. In a static and deterministic setting their data structure implies a linear space and superlinear time preprocessing solution that uses $O(\frac{m}{\alpha}\log\log n)$ and $O(\frac{m}{\alpha}\log\log n+\text{occ})$ query time, respectively.

In this paper, we consider the string indexing problem in the deterministic and packed setting simultaneously, and present a solution that improves all of the above bounds.

4.1.1 Setup and result

We assume a standard unit-cost word RAM with word length $w=\Theta(\log n)$, and a standard instruction set including arithmetic operations, bitwise boolean operations, and shifts. All strings in this paper are over an alphabet Σ of size σ. The *packed representation* of a string A is obtained by storing $\alpha=w/\log\sigma$ characters per word thus representing A in $O(|A|\log\sigma/w)$ words. If A is given in the packed representation we simply say that A is a *packed string*.

Throughout the paper let S be a string of length n. Our goal is to preprocess S into a compact data structure that given a packed pattern string P supports the following queries.

Count(P): Return the number of occurrence of P in S.

Locate(P): Report all occurrences of P in S.

Predecessor(P): Returns the predecessor of P in S, i.e., the lexicographic largest suffix in S that is smaller than P.

We show the following main result.

Theorem 9. *Let S be a string of length n over an alphabet of size σ and let $\alpha=w/\log\sigma$ be the number of characters packed in a word. Given S we can build an index in $O(n)$ deterministic time and space such that given a packed pattern string of length m we can support* Count *and* Predecessor *in time $O(\frac{m}{\alpha}+\log m+\log\log\sigma)$ and* Locate *in time $O(\frac{m}{\alpha}+\log m+\log\log\sigma+\text{occ})$ time.*

Compared to the result of Fischer and Gawrychowski [FG15], Thm 9 is always at least as good and whenever several characters are packed in a word, i.e., $\log \sigma \ll w$, the query times are faster. Compared to the result of Takagi et al. [TISA16], our query time is a factor $\log \log n$ faster.

Technically, our results are obtained by a novel combination of previous techniques. Our general tree decomposition closely follows Fischer and Gawrychowski [FG15], but different ideas are needed to handle packed strings efficiently. We also show how to extend the classic suffix array search algorithm to handle packed strings efficiently.

4.2 Preliminaries

Deterministic hashing and predecessor We use the following results on deterministic hashing and predecessor data structures.

Lemma 6 (Ružić [Ruž08, Theorem 3])**.** *A static linear space dictionary on a set of k keys can be deterministically constructed in time $O(k(\log \log k)^2)$, so that lookups to the dictionary take time $O(1)$.*

Fischer and Gawrychowski [FG15] use the same result for hashing characters. In our context we will apply it for hashing words of packed characters.

Lemma 7 (Fischer and Gawrychowski [FG15, Proposition 7])**.** *A static linear space predecessor data structure on a set of k keys from a universe of size u can be constructed deterministically in $O(k)$ time and $O(k)$ space such that predecessor queries can be answered deterministically in time $O(\log \log u)$.*

Suffix tree The suffix tree T_S of S is the compacted trie over the n suffixes from the string S. We assume that the special character $\$ \notin \Sigma$ is appended to every suffix of S such that each string is ending in a leaf of the tree. The edges are sorted lexicographic from left to right. We say that a leaf *represents* the suffix that is spelled out by concatenating the labels of the edges on the path from the root to the leaf. In the same way an internal node represents a string that is a prefix of at least one of the suffixes. For a node v in T_S, we say that the *subtree* of v is the tree induced by v and all proper descendants of v. We distinguish between implicit and explicit nodes: implicit nodes are conceptual and refer to the original non branching nodes from the trie without compacted paths. Explicit nodes are the branching nodes in the original trie. When we refer to nodes that are not specified as either explicit or implicit, then we are always referring to explicit nodes. The lexicographic ordering of the suffixes represented by the leafs corresponds to the ordering of the leafs from left to right in the compacted trie. For navigating from node to child, each node has a predecessor data structure over the first characters of every edge going to a child. With the predecessor data structure from Lemma 7 navigation from node to child takes $O(\log \log \sigma)$ time and both the space and the construction time of the predecessor data structure is linear in the number of children.

Suffix array Let $S_1, S_2, \ldots, S_n$ be the n suffixes of S from left to right. The suffix array SA_S of S gives the lexicographic ordering of the suffixes such that $S_{\mathsf{SA}_S[i]}$ refers to the ith lexicographic greatest suffix of S. This means that for every $1 < i \leq n$ we have that $S_{\mathsf{SA}_S[i-1]}$ is lexicographic smaller than $S_{\mathsf{SA}_S[i]}$. For simplicity we let $\mathsf{SA}_S[i]$ refer to the suffix $S_{\mathsf{SA}_S[i]}$ and we say that $\mathsf{SA}_S[i]$

represents the suffix $S_{\mathsf{SA}_S[i]}$. Every suffix from S with pattern P as a prefix will be located in a consecutive range of SA_S. This range corresponds to the range of consecutive leafs in the subtree spanned by the explicit or implicit node that represents P in T_S. We can find the range of SA_S where P prefix every suffix by performing binary search twice over SA_S. A naïve binary search takes $O(m \log n)$ time: We maintain the boundaries, L and R, of the current search interval and in each iteration we compare the median string from the range L to R in SA_S, with P, and update L and R accordingly. This can be improved to $O(m + \log n)$ time if we have access to additional arrays storing the value of the longest common prefixes between a selection of strings from SA_S. We construct the suffix array from the suffix tree in $O(n)$ time.

4.3 Deterministic index for packed strings

In this section we describe how to construct and query our deterministic index for packed strings. This structure is the basis for our result in Thm 9. For short patterns where $m < \log_\sigma(n) - 1$ we store tabulated data that enables us to answer queries fast. We construct the tables in $O(n)$ time and space and answer queries in $O(\log\log\sigma + \mathrm{occ})$ time. For long patterns where $m \geq \log_\sigma(n) - 1$ we use a combination of a suffix tree and a suffix array that we construct in $O(n)$ time and space such that queries take $O(m/\alpha + \log\log n + \mathrm{occ})$ time. For $m \geq \log_\sigma(n) - 1$ we have that $\log\log n = \log(\frac{\log n}{\log\sigma}\log\sigma) = \log\log_\sigma n + \log\log\sigma \leq \log(\log_\sigma n - 1) + 1 + \log\log\sigma \leq \log m + 1 + \log\log\sigma$. This gives us a query time of $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ for the deterministic packed index. We need the following connections between T_S and SA_S: For each explicit node t in T_S we store a reference to the range of SA_S that corresponds to the leafs spanned by the subtree of t and for each index in SA_S we store a reference to the corresponding leaf in T_S that represents the same string.

We first describe our word accelerated algorithm for matching patterns in SA_S that we need for answering queries on long patterns. Then we describe how to build and use the data structures for answering queries on short and long patterns.

4.3.1 Packed matching in SA_S

We now show how to word accelerate the suffix array matching algorithm by Manber and Myers [MM93]. They spend $O(m)$ time reading P but by reading α characters in constant time we can reduce this to $O(m/\alpha)$. We let $\mathrm{LCP}(i, j)$ denote the length of the longest common prefix between the suffixes $\mathsf{SA}_S[i]$ and $\mathsf{SA}_S[j]$ and obtain the result in Lemma 8.

Lemma 8. *Given the suffix array SA_S over the packed string S and a data structure for answering the relevant* LCP *queries, we can find the lexicographic predecessor of a packed pattern P of length m in SA_S in $O(m/\alpha + \log n)$ time where α is the number of characters we can pack in a word.*

In the algorithm by Manber and Myers we maintain the left and right boundaries of the current search interval of SA_S denoted by L and R and the longest common prefix between $\mathsf{SA}_S[L]$ and P, and between $\mathsf{SA}_S[R]$ and P, that we denote by l and r, respectively. Initially the search interval is the whole range of SA_S such that $L = 1$ and $R = n$. In an iteration we do as follows: If $l = r$ we start comparing $\mathsf{SA}_S[M]$ with P from index $l+1$ until we find a mismatch and update either L and l, or R and r, depending on whether $\mathsf{SA}_S[M]$ is lexicographic larger or smaller than P. Otherwise, when $l \neq r$, we perform an LCP query that enable us to either half the range of SA_S without reading from P or start comparing $\mathsf{SA}_S[M]$ with P from index $l + 1$ as in the $l = r$ case. When

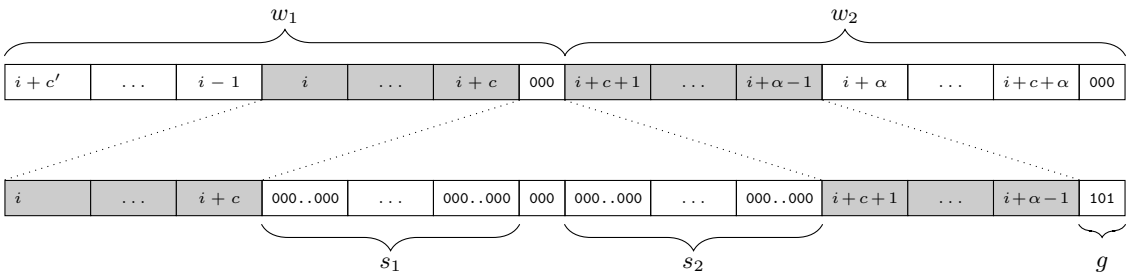

Figure 4.1: Alignment of α characters that extends over a word boundary where $c' = c+1-\alpha$. The relevant part of the lower word w_1 and upper word w_2 is combined with bitwise shifts, a bitwise or and the g bits on the right is set to 0.

$l > r$ there are three cases: If $\text{LCP}(L, M) > l$ then P is lexicographic larger than $\mathsf{SA}_S[M]$ and we set L to M and continue with the next iteration. If $LCP(L, M) < l$ then P is lexicographic smaller than $\mathsf{SA}_S[M]$ and we set R to M and set r to $\text{LCP}(L, M)$ and continue with the next iteration. If $LCP(L, M) = l$ then we compare $\mathsf{SA}_S[M]$ and P from index $l + 1$ until we find a mismatch. Let that mismatch be at index $l+i$. If the mismatch means that P is lexicographic smaller than $\mathsf{SA}_S[M]$ then we set R to M and set r to $l + i - 1$ and continue with the next iteration. If the mismatch means that P is lexicographic larger than $\mathsf{SA}_S[M]$ then we set L to M and set l to $l + i - 1$ and continue with the next iteration. Three symmetrical cases exists when $r > l$.

We generalize their algorithm to work on word packed strings such that we can compare α characters in constant time. In each iteration where we need to read from P we align the next α characters from P and $\mathsf{SA}_S[M]$ such that we can compare them in constant time: Assume that we need to read the range from i to $i + \alpha - 1$ in P. If this range of characters is contained in one word we do not need to align. Otherwise, we extract the relevant parts of the words that contain the range with bitwise shifts and combine them in w_{align} with a bitwise or. See Figure 4.1. We align the α characters from $\mathsf{SA}_S[M]$ in the same way and store them in w'_{align}.

We use a *bitwise exclusive or* operation between w_{align} and w'_{align} to construct a word where the most significant set bit is at a bit position that belong to the mismatching character with the lowest index. We obtain the position of the most significant set bit in constant time with the technique of Fredman and Willard [FW93]. From this we know exactly how many of the next α characters that match and we can increase i accordingly. Since every mismatch encountered result in a halving of the search range of SA_S we can never read more than $O(\log n)$ incomplete chunks. The number of complete chunks we read is bounded by $O(m/\alpha)$. Overall we obtain a $O(m/\alpha + \log n)$ time algorithm for matching in SA_S. This result is summarized in Lemma 8.

4.3.2 Handling short patterns

Now we show how to answer count, locate and lexicographic predecessor queries on short patterns. We store an array containing an index for every possible pattern P where $m < \log_\sigma(n)-1$ and at the index we store a pointer to the deepest node in T_S that prefix P. We call this node d_P. We use d_P as the basis for answering every query on short patterns. We assume that the range in SA_S spanned by d_P goes from l to r. We answer predecessor queries as follows: If P is lexicographic smaller than $\mathsf{SA}_S[0]$ then P has no predecessor in SA_S. Otherwise, we find the predecessor as follows: If d_P is representing P then the predecessor of P is located at index $l - 1$ of SA_S. Otherwise, we

assume that d_P prefix P with i characters and need to decide whether P continues on an edge out of d_P or P deviates from T_S in d_P. We do this by querying the predecessor data structure over the children of d_P with character $i+1$ of P. If this query does not return an edge, then $P[i+1]$ is lexicographic smaller than the first character of every edge out of d_P, and the predecessor of P is the string located at index $l-1$ of SA_S. If this query returns an edge e_{pred} then there are two cases.

Case 1: The first character of e_{pred} is not identical to $P[i+1]$. Then the predecessor of P is the lexicographic largest string in the subtree under e_{pred}.

Case 2: The first character on e_{pred} is identical to $P[i+1]$. In this case, if there exists an edge e'_{pred} out of d_P on the left side of e_{pred}, then the predecessor of P is the lexicographic largest string in the subtree under e'_{pred} and otherwise the predecessor is the string at index $l-1$ of SA_S. We report the node in T_S that represents the predecessor of P.

We let e_{pred} be defined as above and answer count queries as follows: If d_P represents P we return the number of leafs spanned by d_P in T_S. If P instead continues and ends on e_{pred} we report the number of leafs spanned by the subtree below e_{pred}. We answer locate queries in the same way but instead of reporting the range we report the strings in the range.

We find d_P in $O(1)$ time and e_{pred} in $O(\log\log\sigma)$ time. In total we answer predecessor and count queries in $O(\log\log\sigma)$ time and locate queries in $O(\log\log\sigma + \mathrm{occ})$ time

Since $m < \log_\sigma(n) - 1$ there exists $\sigma + \sigma^2 + \ldots + \sigma^{\lfloor \log_\sigma(n)-1 \rfloor} \leq \sigma^{\lfloor \log_\sigma(n) \rfloor} \leq \sigma^{\log_\sigma n} = n$ short patterns and we compute them in $O(n)$ time by performing a preorder traversal of T_S bounded to depth $\log_\sigma(n) - 1$. Let d_P be the node we are currently visiting and let d_{next} be the node we visit next. When we visit d_P we fill the tabulation array for every string that is lexicographic larger or equal to the string represented by d_P and lexicographic smaller than the string represented by d_{next}. We fill each of these indices with a pointer to d_P since d_P is the deepest node in T_S that represents a string that prefix these strings. We can store the tabulation array in $O(n)$ space.

4.3.3 Handling long patterns

Now we show how to answer count, locate and lexicographic predecessor queries on long patterns. We first give an overview of our solution followed by a detailed description of the individual parts. In T_S we distinguish between *light* and *heavy* nodes. If a subtree under a node spans at least $\log^2\log n$ leafs, we call the node heavy, otherwise we call it light. A node is a heavy branching node if it has at least two heavy children and all the heavy nodes constitutes a subtree that we call the heavy tree. We decompose the heavy tree into micro trees of height α and we augment every micro tree with a data structure that enables navigation from root to leaf in constant time. For micro trees containing a heavy branching node we do this with deterministic hashing and for micro trees without a heavy branching node we just compare the relevant part of P with the one unique path of the heavy tree that goes through the micro tree. To avoid navigating the light nodes we in each light node store a pointer to the range of SA_S that the node spans. We construct two predecessor data structures for each micro tree: The *light predecessor* structure over the strings represented by the light nodes that are connected to the heavy nodes in the micro tree and the *heavy predecessor* structure over the heavy nodes in the micro tree. We answer queries on P as follows: We traverse the heavy tree in chunks of α characters until we are unable to traverse a complete micro tree. This means that P either continues in a light node, ends in the micro tree or deviates from T_S in the micro tree. We can decide if P continues in a light node with the light predecessor structure and if this is the case we answer the query with the packed matching algorithm on the range of SA_S spanned by the light node. Otherwise, we use the heavy predecessor structure for finding d_P in the

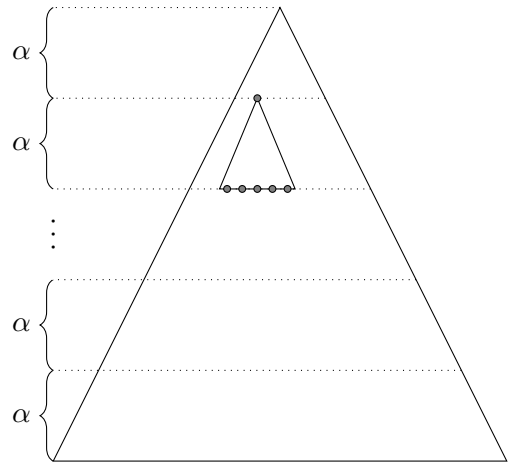

Figure 4.2: The decomposition of HT_S in micro trees of height α. One micro tree is shown with the root at string depth α and the boundary nodes at string depth 2α

micro tree and use d_P for answering the query as in section 4.3.2. The following sections describes in more detail how we build our data structure and answer queries and gives a time and space .

4.3.3.1 Data structure

This section describes our data structure in details. If a subtree under a node in T_S spans at least $\log^2 \log n$ leafs, we call the node heavy. The heavy tree HT_S is the induced subgraph of all the the heavy nodes in T_S. We decompose HT_S into *micro trees* of string depth α. A node, explicit or implicit, is a boundary node if its string depth is a multiple of α. Except for the original root and leafs of HT_S, each boundary node belongs in two micro trees i.e., a boundary node at depth $d\alpha$ is root in a micro tree that starts at string depth $d\alpha$ and is a leaf in a micro tree that starts at string depth $(d-1)\alpha$. Figure 4.2 shows the decomposition of HT_S into micro trees of string depth α.

We augment every micro tree with information that enables us to navigate from root to leaf in constant time. To avoid using too much space we promote only some of the implicit boundary nodes to explicit nodes. We distinguish between three kinds of micro trees:

- **Type 1.** At least one heavy branching node exists in the micro tree: We promote the root and leafs to explicit nodes and use deterministic hashing to navigate the micro tree from root to leaf. Because the micro tree is of height α, each of the strings represented by the leafs in the micro tree fits in a word and can be used as a key for hashing. We say that the root is a hashing node and the leafs are hashed nodes. We will postpone the analysis of time and space used by the micro trees that use hashing for navigation.

- **Type 2.** No heavy branching node exists in the micro tree: When the micro tree does not contain a heavy branching node, the micro tree is simply a path from root to leaf. Here we distinguish between two cases:

  - **Type 2a.** The micro tree contains a non branching heavy node:
    We promote the root and leaf to explicit nodes. Navigating from root to leaf takes constant time by comparing the string represented by the leaf with the appropriate part

of P. We charge the space increase from the promotion of the root and leaf to the non branching heavy node. Since there are at most n non branching heavy nodes we never promote more than $2n$ implicit nodes from type 2a micro trees.

- **Type 2b.** The micro tree does not contain a heavy node: If the root is a boundary node where the micro tree above contains a heavy node we promote the root to an explicit node and store a pointer to the root of the nearest micro tree below that contains a heavy node. The path from root to root corresponds to a substring in S and we navigate by comparing this string to the appropriate part of P. We charge the space increase from the promotion of the root to the heavy node descendant. Since we have at most n heavy nodes we promote no more than n implicit nodes from type 2b micro trees. We ignore every micro tree where the micro tree above does not contain a heavy node.

We say that a node in T_S is a heavy leaf if it is a heavy node with no heavy children. We want to bound the number of heavy branching nodes and heavy leafs. Every heavy leaf spans at least $\log^2 \log n$ leafs of T_S. This means we can have at most $n/\log^2 \log n$ heavy leafs in T_S. Since we have at most one branching heavy node per heavy leafs the number of heavy branching nodes is at most $n/\log^2 \log n$.

We want to bound the number of implicit nodes that are promoted to explicit hashed nodes. This number is critical for constructing all hash functions in $O(n)$ time. We bound the number of promoted hashed nodes by associating each with the nearest descendant that is either a heavy branching node or a heavy leaf: Let l be a promoted hashed node in a micro tree that contain a heavy branching node h. Then every promoted hashed node above l is associated with h or a node above h in the tree. Hence, no other promoted node can be associated with the first encountered heavy branching or leaf node below l. Since we have at most $O(n/\log^2 \log n)$ heavy branching and heavy leaf nodes we also have at most $O(n/\log^2 \log n)$ implicit nodes that are promoted to explicit hashed nodes.

With deterministic hashing from Lemma 6 the total time for constructing the explicit hashing nodes are:

$$O\left(\sum_{h\in H} |h| \log^2 \log |h|\right) = O\left(\sum_{h\in H} |h| \log^2 \log(n/\log^2 \log n)\right)$$

$$= O\left(\log^2 \log(n/\log^2 \log n) \cdot \sum_{h\in H} |h|\right) = O\left(\log^2 \log(n/\log^2 \log n) \frac{n}{\log^2 \log n}\right) = O(n)$$

Here H is the set of all the hash functions and we bound the elements in every hash function h to $n/\log^2 \log n$. Summing the elements of every hash function is bounded by the maximum number of promoted nodes, i.e. $O(n/\log^2 \log n)$. To conclude, we spend linear time constructing the hash functions in the micro trees that contain a heavy branching node.

We associate two predecessor data structures with each micro tree that contains a heavy node: The first predecessor structure contains every light node that is a child of a heavy node in the micro tree. We call this predecessor data structure for the *light predecessor structure* of the micro tree. The key for each light node is the string on the path from the root of the micro tree to the node itself padded with character \$ such that every string has length α. These keys are ordered lexicographic in the predecessor data structure and a successful query yields a pointer to the node. The second predecessor structure is similar to the first but contains every heavy node in the micro tree. We call

this predecessor structure for the *heavy predecessor structure*. We use Lemma 7 for the predecessor structures. The total size of every light and heavy predecessor structures is $O(n)$ and a query in both take $O(\log\log n)$ because the universe is of size $(\sigma+1)^\alpha$.

For each light node that are a child of a heavy node we additionally store pointers to the range of SA_S that corresponds to the leafs in T_S that the light node spans.

4.3.3.2 Answering queries

We answer queries on long patterns as follows. First we search for the deepest micro tree in HT_S where the root prefix P. We do this by navigating the heavy tree in chunks of α characters starting from the root. Assuming that we have already matched a prefix of P consisting of i chunks of α characters we need to show how to match the $(i+1)$th chunk: If the micro tree is of type 1 and P has length at least $(i+1)\alpha$, we try to hash the substring $P[i\alpha,(i+1)\alpha]$. If we obtain a node v from the hash function we continue matching chunk $P[(i+1)\alpha,(i+2)\alpha]$ from v. If the micro tree is of type 2 we compare α sized chunks of P with the string on the unique path from root to the first micro tree with an explicit root and continue matching from here. We have found the deepest micro tree where the root prefix P when we are unable to match a complete chunk of α characters or are unable reach a micro tree with an explicit root. From this micro tree we need to decide whether the query is answered by searching SA_S from a light node or answered by finding d_P in the micro tree, where d_P is defined as in Section 4.3.2, i.e. the deepest node in T_S that prefix P. We check if P continues in a light node by querying the light predecessor structure of the micro tree with the next unmatched α characters from P and pad with character \$ if less than α characters remain unmatched in P. If the light node returned by the query represents a string that prefix P we answer the query by searching the range of SA_S spanned by the light node with the packed matching algorithm.

When P does not continue in a light node we instead find and use d_P for answering the query: If the micro tree is of type 2b or the root of the micro tree represents P then d_P is the root of the micro tree. Otherwise, we find d_P with a technique, very similar to a technique used by Fredman and Willard [FW93], that query the heavy predecessor structure three times as follows: We call the remaining part of P, padded to length α with character \$, for p_0. We first query the predecessor structure with p_0 which yields a node that represents a string n_0. We then construct a string, p_1, that consists of the longest common prefix of p_0 and n_0, and as above, padded to length α. We query the predecessor structure with p_1 which yield a new node that represents a string n_1. We then construct a string, p_2, that consists of the longest common prefix of p_0 and n_1, again padded to length α. At last, we query the predecessor structure with p_2 which returns d_P. Given d_P, we answer count, locate and lexicographic predecessor queries exactly as we did in section 4.3.2.

Now we prove the correctness of our queries. First we prove that if P continues in a light node then the query in the light predecessor structure returns that light node: Assume that P goes through the light node l_P that has a heavy parent in the micro tree T_p and that we query the light predecessor structure with the string Q_α. Let L_{pred} be the string that represents l_P in the light predecessor structure. Since P goes through l_P then L_{pred} is identical or lexicographic smaller than Q_α. Let L'_{pred} be the successor of L_{pred} in the light predecessor structure. Since L_{pred} is lexicographic smaller than L'_{pred} and has a longer common prefix with Q_α than L'_{pred} has with Q_α, then L'_{pred} must be lexicographic larger than Q_α. Since Q_α is identical or lexicographic larger than L_{pred} and lexicographic smaller than L'_{pred}, a query on Q_α in the light predecessor structure will return l_P.

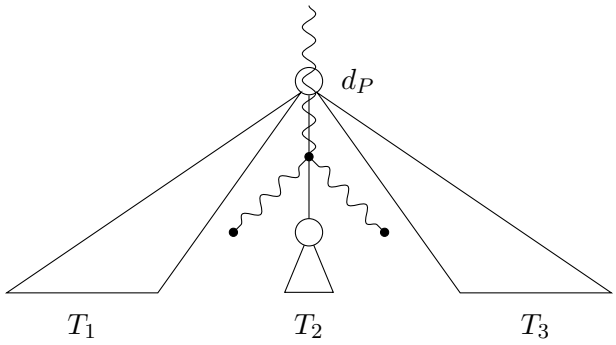

Figure 4.3: Searching for a prefix of P in HT_S

We now prove that the queries in the heavy predecessor structure always returns d_P: Because P is not prefixed by a leaf of the micro tree or a light node from the light predecessor structure we know that d_P is a heavy node in the micro trie. In Figure 4.3, d_P is depicted and P either ends on or deviates from the edge e that leads to the tree T_2. The trees T_1, T_2 and T_3 combined with d_P and the edge e constitutes the subtree of d_P. If P deviates to the left or ends on e then P is lexicographic smaller than every string represented in T_2. If P deviates to the right then P is lexicographic larger than every string represented in T_2. Assume that P deviates to the right on e. Then the query to the heavy predecessor structure with pattern p_0 will yield n_0 that represents the lexicographic largest string in T_2. The pattern p_1 will then be represented by the implicit node from where P deviates from e. The pattern p_1 is lexicographic smaller than every string represented in T_2 and a query will yield n_2 as the lexicographic largest node in T_1 or, if T_1 is empty, the node d_P. Either way, the query on p_2 will yield the node d_P. We can make similar arguments for the other cases where P ends on e, deviates left from e, ends at d_P or goes through d_P without following e.

The following gives an analysis of the running time of our queries. We spend at most $O(m/\alpha)$ time traversing the heavy tree. Both predecessor structures contains strings over a universe of size n such that a query takes $O(\log\log n)$ time using Lemma 7. Each light node spans at most $\log^2\log n$ leafs which corresponds to an interval of length $\log^2\log n$ in SA_S that we search in $O(m/\alpha + \log\log\log n)$ time with the word accelerated algorithm for matching in SA_S. Overall, we spend $O(m/\alpha + \log\log n)$ time for answering count and lexicographic predecessor queries and $O(m/\alpha + \log\log n + \mathrm{occ})$ time for answering locate queries. Since we only query this data structure for patterns where $m \geq \log_\sigma(n) - 1$ we have that $\log\log n = \log(\frac{\log n}{\log\sigma}\log\sigma) = \log\log_\sigma(n) + \log\log(\sigma) \leq \log(\log_\sigma(n)-1)+1+\log\log(\sigma) \leq \log(m)+1+\log\log(\sigma)$, such that we answer count and lexicographic predecessor queries in $O(m/\alpha + \log m + \log\log\sigma)$ time and locate queries in $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ time. Combined with our solution for patterns where $m < \log_\sigma(n) - 1$, that answer the queries in $O(\log\log\sigma)$ and $O(\log\log\sigma + \mathrm{occ})$ time, respectively, we can for patterns of *any* length answer count and lexicographic predecessor queries in $O(m/\alpha + \log m + \log\log\sigma)$ time and locate queries in $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ time. This is our main result which is summarized in Thm 9.

Chapter 5

Dynamic Partial Sums in Constant Time and Succinct Space with the Ultra Wide Word-RAM Model

Dynamic Partial Sums in Constant Time and Succinct Space with the Ultra Wide Word-RAM Model

Philip Bille* phbi@dtu.dk
Inge Li Gørtz* inge@dtu.dk
Frederik Rye Skjoldjensen† fskj@dtu.dk

Technical University of Denmark
{phbi,inge,fskj}@dtu.dk

Abstract

The dynamic partial sums problem is to dynamically maintain an array of n integers while supporting efficient access, update and partial sums queries. This classic problem, and its variations, are very well studied in many different computational models [Fre82, FS89, Fen94, HSS11, HR03, HRS96, RRR01, PD04]. We solve the partial sums problem in the *ultra wide word-RAM model*, recently introduced by Farzan et al. [FLONS15], where we, in constant time, are allowed to manipulate words of size w^2 and access w memory locations. Farzan et al. [FLONS15] additionally gave a solution to the dynamic partial sums problem by simulating the RAMBO model to obtain a result by Brodnik et al. [BKMN06]. In this paper we present an improved solution to the dynamic partial sums problem in the ultra wide word-RAM model that supports all operations in either constant or $O(\log \log n)$ time, depending on whether we allow multiplication, and succinct space. We pose as an open problem whether it is possible in the ultra wide word-RAM model to additionally support the classic select operation in constant time.

5.1 Introduction

The *dynamic partial sums problem* is to maintain an integer array A of length n while efficiently supporting Access queries (return integer at specific index), Update queries (increment or decrement a specific index) and Sum queries (return the sum of every integer up to a given index). The problem is very well studied in many different computational models. Initially, two different versions of the problem was studied by Fredman [Fre82]: An algebraic version, where the elements in A are elements from a group and the complexity is measured as the number of composed operations needed to answer the query, and a version where only integers modulo 2 is considered and time complexity is measured as the number of memory accesses and memory changes. Dietz [Die89] solves the dynamic partial sums problem in the RAM-model with a word size of $O(\log n)$-bits and $O(\log n / \log \log n)$ time complexity for all operations under the assumption that Update are constrained to values representable with $O(\log \log n)$ bits. The problem was further studied by many authors [FS89,Fen94,HSS11,HR03,HRS96,RRR01] and in particular, Demaine and Pătraşcu [PD04]

*Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178)
†Supported by the Danish Research Council (DFF – 1323-00178)

shows a matching upper and lower bound for the partial sums problem with the additional query Select. Demaine and Pătraşcu further gives an elaborate exposition of previous work. Fredman and Saks [FS89] claimed that solution where each operation takes constant time is possible in the *RAM model with bit overlap*(RAMBO). In this model, memory words can overlap on a subset of their bits. This claim was investigated by Brodnik et al. [BKMN06] that gave a constant time solution in the RAMBO model with the restriction that $\lg n \lg U < w$, where U is the size of the universe and w is the word size. Farzan et al. [FLONS15] introduced the ultra wide word-RAM model and showed that it can simulate any algorithm of the RAMBO model with a space overhead. By this simulation they transfer the result by Brodnik et al. [BKMN06] to the ultra wide word-RAM model. The ultra wide word-RAM model extends the standard unit cost word-RAM model with strong capabilities for manipulating wide bitstrings and parallel addressing. Farzan et al. [FLONS15] supports all operations of the dynamic partial sums problem in $O(\tau + 1)$ time an $O(\log(U) \cdot U^{\log(n)/2^\tau} + n \log(u) \log(n) w)$ space, where U is the size of the universe of the elements that can be stored in A and τ is a trade-off parameter such that $1 \leq \tau \leq \log(n)$. This is still under the constrain that $\lg n \lg U < w$. The additive term on the left stems from the space overhead of the simulation of the RAMBO model. They do not present any solution specifically designed for the ultra wide word-ram model.

Both Brodnik et al. [BKMN06] and Farzan et al. [FLONS15] obtain their results without using word level parallelism for summing integers that are packed in a single word and instead use a precomputed table.

In this paper we present a solution to the dynamic partial sums problem in the ultra wide word-RAM model that supports all operations in constant time and succinct space. Additionally, for the ultra wide word-RAM model without multiplication, we support all operations in $O(\log \log n)$ time and succinct space.

5.1.1 Setup and Result

As computational model we work the *ultra wide word-RAM model* (UW-RAM) that was recently proposed by Farzan et al. [FLONS15]. The UW-RAM is a unit-cost word RAM with extended capabilities for bit manipulation and addressing.

Throughout the paper let A be an array of n w-bit integers. Our goal is to maintain A under the following operations:

Access(i): Return $A[i]$.

Sum(i): Returns the sum of every index smaller than or equal to i, i.e. $\sum_{j=1}^{i} A[i]$.

Update(i, v): Increment index i of A with value v, i.e $A[i] := A[i] + v$, where v is any value representable in two's complement representation with w-bits.

We show the following result:

Theorem 10. *Let A be an array of n positive w-bit integers. In the ultra wide word-RAM model with multiplication we can build a data structure in $O(n)$ time and $n + O(\log n)$ words of space that support Access, Sum and Update operations in $O(1)$ time.*

For a UW-RAM *without* support for multiplication we show the following result:

Lemma 9. *Let A be an array of n positive w-bit integers. In the ultra wide word-RAM model without multiplication we can build a data structure in $O(n)$ time and $n + O(\log n)$ words of space that support* Access, Sum *and* Update *operations in $O(\log \log n)$ time.*

5.2 The Ultra Wide Word-RAM Model

The UW-RAM is a standard unit-cost RAM with extended capabilities for working with wide words and parallel addressing. The UW-RAM has a word length of $w \geq \log n$ and a standard instruction set including uarithmetic operations, bitwise boolean operations and shifts that operate on w-bit words. Additionally, every instruction can also manipulate *wide words* of w^2-bits and load w^2 bits from memory, both in constant time. We distinguish between *standard* words of w bits and *wide* words of w^2 bits. We will also consider a variation of the model where we exclude multiplication from the instruction set.

We use the notation by Farzan et al. [FLONS15] for indexing words: Let W be a wide word of w^2 bits and let $W[i]$ denote the i-th bit of W. We let $W[0]$ denote the least significant bit of W and let $W[i \dots j]$ denote the contigous subword of w bits that corresponds to the bitstring $W[i]W[i+1] \dots W[j]$. Let W_j denote the j-th contiguous subword of W that consists of subword $W[jw \dots (j+1)w - 1]$ for $0 \leq j < w$ and let $W_j[i]$ denote the i-th bit of W_j. The UW-RAM only makes a distinction between wide words and standard words when addressing memory. For all other operations, a wide word is treated exactly as a standard word.

The UW-RAM supports the standard addressing modes of the unit-cost word RAM but extends it capabilities with the following constant time addressing modes. Let M denote the addressable memory. A *simple* read takes an index i of M and loads a wide word W such that $W_j = M[i+j]$ for $0 \leq j \leq w - 1$, i.e, a simple read loads w consecutive standard words into a wide word. A *scattered* read takes a wide word W^a and loads a wide word W^l such that $W^l_j = M[W^a_j]$ for $0 \leq j \leq w-1$, i.e, a scattered read loads w standard words from w different addresses given by $W^a_0, W^a_1, \dots, W^a_{w-1}$. We define simple and scattered writes similarly. Farzan et al. [FLONS15] further defines extends the UW-RAM with a *spread* and *compress* operation. These are not standard word RAM operations but can be implemented in a constant depth circuit model such as AC^0. The compress operations on a wide word W, copies bit $W[0]_j$ into $W'[j]_0$ for all $0 \leq j < w$. The spread operation is the inverse, and copies bit $W[j]_0$ into $W'[0]_j$.

5.3 Preliminaries

We use the Fenwick tree introduced by Fenwick [Fen94] as the data structure for representing the array A.

Fenwick tree The Fenwick tree by Fenwick [Fen94] is a data structure for maintaining an array A of n w-bit integers that supports Access, Sum and Update queries in $O(\log n)$ time on the standard word-RAM. The data structure is stored implicitly in an array A' of n words. Index i of A' stores the value $\sum_{j=i-\Delta(i)+1}^{i} A[j]$ where $\Delta(i)$ is the largest power of two that divides i. This value, $\Delta(i)$, is also the value represented by the least significant set bit of i.

We define the functions $\mathtt{add}_{\mathtt{LSB}}(i)$ to be $i + \Delta(i)$ and $\mathtt{sub}_{\mathtt{LSB}}(i)$ to be $i - \Delta(i)$, such that the functions return i incremented and decremented with the value of the least significant bit of i. For conve-

nience we define repeated application of the functions such that $\texttt{add}_{\texttt{LSB}}{}^k(i) = \texttt{add}_{\texttt{LSB}}{}^{k-1}(\texttt{add}_{\texttt{LSB}}(i))$ and $\texttt{add}_{\texttt{LSB}}{}^1(i) = \texttt{add}_{\texttt{LSB}}(i)$ and similar for $\texttt{sub}_{\texttt{LSB}}{}^k(i)$.

We answer $\mathsf{Sum}(i)$ by summing $A'[i], A'[\texttt{sub}_{\texttt{LSB}}(i)], A'[\texttt{sub}_{\texttt{LSB}}{}^2(i)], \ldots, A'[\texttt{sub}_{\texttt{LSB}}{}^{s-1}(i)]$, where s is the number of set bits in the binary representation of i. As an example, if $n = 32$ and for $\mathsf{Sum}(13)$ we sum $A'[001101_b], A'[001100_b]$ and $A'[001000_b]$.

We perform an $\mathsf{Update}(i, v)$ by adding the value v to $A'[i]$, $A'[\texttt{add}_{\texttt{LSB}}(i)]$, $A'[\texttt{add}_{\texttt{LSB}}{}^2(i)]$, ..., $A'[\texttt{add}_{\texttt{LSB}}{}^{\overline{s}-1}(i)]$, where $\overline{s}$ is the number of non set bits in the binary representation of i that represents a value greater than the least significant set bit in i. As an example, if $n = 32$ and we need to perform an $\mathsf{Update}(18, v)$ we add v to $A'[010010_b], A'[010100_b], A'[011000_b]$ and $A'[100000_b]$. An $\mathsf{Access}(i)$ operation is simply answered by $\mathsf{Sum}(i) - \mathsf{Sum}(i-1)$.

We calculate the indices produced by $\texttt{add}_{\texttt{LSB}}(i)$ and $\texttt{sub}_{\texttt{LSB}}(i)$ by finding the least significant set bit of i. Let i^- be the two's complement representation of $-i$. Performing a bitwise `and` operation between i and i^- will produce a bitstring i_Δ with a single set bit at the position that corresponds to $\Delta(i)$. To produce $\texttt{add}_{\texttt{LSB}}(i)$ and $\texttt{sub}_{\texttt{LSB}}(i)$ we then either add or subtract the value i_Δ. Given an index i, we can make no more than $O(\log n)$ repeated applications since every index of A' can be represented with $O(\log n)$ bits. This means that the Fenwick tree can support the operations Access, Sum and Update in $O(\log n)$ time.

Word Parallel Summing and Summing in the UW-RAM Let W be a wide word containing w-bit integers in the subwords $W_0, W_1, \ldots, W_{w-1}$. To speed up the queries on the Fenwick tree in the UW-RAM, we need methods for efficiently summing the integers of W and a method for adding a w-bit value v to each integer of W. We describe two methods for doing this depending on whether multiplication is allowed in the model.

Without multiplication For summing the integers, let W^u be the upper half of the bits of W shifted down $w^2/2$ positions, and let W^l be the lower half of the bits of W. Adding together W^u and W^l gives us a wide word W' where W'_j contains the sum of $W_{j+w/2}$ and W_j, for $1 \leq j \leq w/2$. We continue recursively for $\log w$ steps until the sum of every integer is contained in the lowest subword. With this approach we use $O(\log w)$ time for summing the w integers.

For adding v to the integers, let W^k be the wide word where $W^k_j = v$ for all $1 \leq j \leq k$. We construct W^k in constant time by shifting $W^{k/2}$ up $k/2$ positions and then adding $W^{k/2}$, i.e $(W^{k/2} << k/2) + W^{k/2}$. In this way we construct W^w in $O(\log w)$ time.

With multiplication For summing the integer, we multiply W with the bit pattern W' where $W'_j[1] = 1$ for every $0 \leq j < w$ such that the value of the sum is located in subword w of the resulting wide word. This takes constant time.

For adding v to the integers, we multiply W' with W'' where $W''_1 = v$. This puts the value of v in every subword of the result that we add to W. This also takes constant time.

5.4 Dynamic Partial Sums in UW-RAM

We speed up the operations of the Fenwick tree in the UW-RAM such that all operations take $O(1)$ time. The Fenwick tree is laid out in memory exactly as before and we speed up the queries solely by taking advantage of the wide word size and the parallel addressing capabilities of the UW-RAM. The general idea for performing both the Access and Update query is to calculate, with word level

parallelism, all indices of A' that is needed for the query and then either read or update the indices with parallel addressing.

To perform a $\mathsf{Sum}(i)$ we need to add together $A'[i]$, $A'[\mathtt{sub}_{\mathtt{LSB}}(i)]$, $A'[\mathtt{sub}_{\mathtt{LSB}}{}^2(i)]$, ..., $A'[\mathtt{sub}_{\mathtt{LSB}}{}^{s-1}(i)]$. For this we need to calculate all the indices in parallel: In the wide word W^0 we have the w-bit representation of i at every subword W^0_j for $0 \leq j < w$ and M^0 is a wide word mask pattern where M^0_j has the $j-1$ least significant bits set to $\mathtt{0}$ and the $w-(j-1)$ most significant bits set to $\mathtt{1}$. We compute $W^1 := W^0 \,\&\, M^0$ such that each W^1_j contains an index of A' that is part of the sum. Some of the indices in W^1 might be duplicates so we need to sum only a subset that we select with a bit mask. We select the index contained in W^1_j if bit j of the bit representation of i is set to $\mathtt{1}$. We let M^1 be this mask pattern where every bit of M^1_j is set to $\mathtt{1}$ if bit j of i is $\mathtt{1}$ and set to $\mathtt{0}$ if bit j of i is $\mathtt{0}$. We mask out the correct indices of W^1 by computing $W^2 := W^1 \,\&\, M^1$ and dereference W^2 such that we obtain a wide word W^3 where W^3_j contains the value of $A'[W^2_j]$. The result of the $\mathsf{Sum}(i)$ query is then the result of a parallel sum operation of W^3.

To perform an $\mathsf{Update}(i, v)$ we need to add the value v to $A'[i]$, $A'[\mathtt{add}_{\mathtt{LSB}}(i)]$, $A'[\mathtt{add}_{\mathtt{LSB}}{}^2(i)]$, ..., $A'[\mathtt{add}_{\mathtt{LSB}}{}^{\bar{s}-1}(i)]$. We do as for the Sum query but make the following changes: We calculate $W^1 := (W^0 \,\&\, M^0) + P^+$ such that we now also add the wide word P^+ where $P^+_j[j] := \mathtt{1}$ and all other bits are set to $\mathtt{0}$. Let l be the the bit position of the least significant set bit of i. Before using M^1 to calculate W^2 we set all bits in M^1_j to $\mathtt{0}$, for $0 \leq j < l$, and negate every bit in M^1_k, for $l \leq k < w$. Instead of performing a parallel sum on W^3 we parallel add v to W^3 and with the addresses stored in W^2, write the result back to the right indices of A'.

We assume that M^0 and P^+ are constructed beforehand an if we allow multiplication we can construct W^0 and M^1 in constant time and sum every value of W^3 in constant time. We construct W^0 with a parallel add and M^1 with a spread operation and a parallel add operation. This gives Theorem 10. Without multiplication we need $O(\log\log n)$ time for constructing W^0, M^1 and summing W^3. This gives Lemma 9.

As an example, consider the $\mathsf{Sum}(5)$ query on the following Fenwick tree:

i	1	2	3	4	5	6	7	8
$A =$	12	8	2	7	9	4	3	6
$A' =$	12	20	2	29	9	13	3	51

We need to calculate the indices to access and we start with calculating $W^1 := W^0 \,\&\, M^0$:

i	1	2	3	4	5	6	7	8
$W^0 =$	0000101	0000101	0000101	0000101	0000101	0000101	0000101	0000101
$M^0 =$	1111111	1111110	1111100	1111000	1110000	1100000	1000000	0000000
$W^1 =$	0000101	0000100	0000100	0000000	0000000	0000000	0000000	0000000

We then need to select the correct indicies with the operation $W^2 := W^1 \,\&\, M^1$

i	1	2	3	4	5	6	7	8
$W^1 =$	0000101	0000100	0000100	0000000	0000000	0000000	0000000	0000000
$M^1 =$	1111111	0000000	1111111	0000000	0000000	0000000	0000000	0000000
$W^2 =$	0000101	0000000	0000100	0000000	0000000	0000000	0000000	0000000

We access these indices with a parallel address and parallel sum the values to get 38.

5.4.1 Discussion and open problems

It is natural to extend the dynamic partial sums problem to also support the operation $\mathsf{Select}(v)$, that returns index i of A where $\mathsf{Sum}(i-1) < v \leq \mathsf{Sum}(i)$. For Fenwick trees on the standard unit-cost word RAM we can support the Select operation in $O(\log n)$ time with a binary search over the indices of A'. When answering $\mathsf{Access}(i)$, $\mathsf{Sum}(i)$ and $\mathsf{Update}(i, v)$ operations for Fenwick trees on UW-RAM we can calculate every index of A' that we need to access directly from the index i. This is not the case for the Select operation. Here parallel calculation of the indices is prevented because each index accessed is dependent on previously accessed indices. We pose as an open problem whether it is possible to maintain an array of n w-bit integers on the UW-RAM while supporting Access, Sum, Update and Select operations in constant time.

Bibliography

[ABR00] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proc. 11th SODA*, pages 819–828, 2000.

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[ALLS07] Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM TALG*, 3(2):19, 2007.

[BBPV10] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *European Symposium on Algorithms*, pages 427–438. Springer, 2010.

[Bel12] Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Disc. Algorithms*, 14:91–106, 2012.

[BFC08] Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoret. Comput. Sci.*, 409:486 – 496, 2008.

[BGVV14] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.

[BGW00] Adam L Buchsbaum, Michael T Goodrich, and Jeffery R Westbrook. Range searching over tree cross products. In *European symposium on algorithms*, pages 120–131. Springer, 2000.

[Bil11] Philip Bille. Fast searching in packed strings. *Journal of Discrete Algorithms*, 9(1):49–56, 2011.

[BIST03] Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In *Proc. 28th MFCS*, pages 208–217, 2003.

[BKBB+14] Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theoret. Comput. Sci.*, 525:111–129, 2014.

[BKMN06] Andrej Brodnik, Johan Karlsson, J Ian Munro, and Andreas Nilsson. An o (1) solution to the prefix sum problem on a specialized memory architecture. In *Fourth IFIP International Conference on Theoretical Computer Science-TCS 2006*, pages 103–114. Springer, 2006.

[Bro95] Andrej Brodnick. *Searching in constant time and minimum space.* PhD thesis, University of Waterloo, 1995.

[BY91] Ricardo A. Baeza-Yates. Searching subsequences. *Theoret. Comput. Sci.*, 78(2):363–376, 1991.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th STOC*, pages 91–100, 2004.

[CKL06] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *Automata, Languages and Programming*, pages 358–369. Springer, 2006.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, second edition.* MIT Press, 2001.

[CMT03] Maxime Crochemore, Borivoj Melichar, and Zdeněk Troníček. Directed acyclic subsequence graph: Overview. *J. Disc. Algorithms*, 1(3-4):255–280, 2003.

[COM+12] BG Chern, Idoia Ochoa, Alexandros Manolakos, Albert No, Kartik Venkat, and Tsachy Weissman. Reference based genome compression. In *IEEE ITW*, pages 427–431, 2012.

[CT99] Maxime Crochemore and Zdeněk Troníček. Directed acyclic subsequence graph for multiple texts. *Technical Repport IGM-99-13, Institut Gaspard-Monge*, 1999.

[Die89] Paul F Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, pages 39–46, 1989.

[DJSS14] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative Lempel–Ziv self-index for similar sequences. *TCS*, 532:14–30, 2014.

[FCFM00] Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[Fen94] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[FFMR10] Effat Farhana, Jannatul Ferdous, Tanaeem Moosa, and M Sohel Rahman. Finite automata based algorithms for the generalized constrained longest common subsequence problems. In *Proc. 17th SPIRE*, pages 243–249, 2010.

[FG15] Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching*, pages 160–171. Springer, 2015.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FLONS15] Arash Farzan, Alejandro López-Ortiz, Patrick K Nicholson, and Alejandro Salinger. Algorithms in the ultra-wide word model. In *International Conference on Theory and Applications of Models of Computation*, pages 335–346. Springer, 2015.

[FM96] Martin Farach and S Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 130–140. Springer, 1996.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[FMMN04] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Succinct representation of sequences. Technical report, Technical Report TR/DCC-2004-5 (Aug.), Department of Computer Science, University of Chile, Chile, 2004.

[Fre82] Michael L Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM (JACM)*, 29(1):250–260, 1982.

[FS89] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

[FV07] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *TCS*, 372(1):115 – 121, 2007.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.

[GGLP15] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space. In *Proc. 26th SODA*, pages 769–775, 2015.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[GLN14] Paweł Gawrychowski, Moshe Lewenstein, and Patrick K Nicholson. Weighted ancestors in suffix trees. In *Proc. 22nd ESA*, pages 455–466. 2014.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge, 1997.

[Han93] Christophe Hancart. On simon's string searching algorithm. *Information Processing Letters*, 47(2):95–99, 1993.

[HL07] Christopher L Hayes and Yan Luo. Dpico: a high speed deep packet inspection engine using compact finite automata. In *Proc. 3rd ANCS*, pages 195–203, 2007.

[HPZ11] Christopher Hoobin, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

[HR03] Thore Husfeldt and Theis Rauhe. New lower bound techniques for dynamic partial sums and related problems. *SIAM J. Comput.*, 32(3):736–753, 2003.

[HRS96] Thore Husfeldt, Theis Rauhe, and Søren Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. In *Proc. 5th SWAT*, pages 198–211, 1996.

[HSS11] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *TCS*, 412(39):5176–5186, 2011.

[HSTA00] Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Online construction of subsequence automata for multiple texts. In *Proc. 7th SPIRE*, pages 146–152, 2000.

[HT84] Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[Huf54] David A Huffman. The synthesis of sequential switching circuits. *Journal of the franklin Institute*, 257(3):161–190, 1954.

[JSS12] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. CRAM: Compressed random access memory. In *Proc. 39th ICALP*, pages 510–521. 2012.

[KDY+06] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. 12th SIGCOMM*, pages 339–350, 2006.

[KJHMP77] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KPZ10] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.

[KPZ11] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proc. 34th ACSC*, pages 91–98, 2011.

[KR78] Brian Kernighan and Dennis Ritchie. *The C Programming Language (1st Ed.)*. Prentice-Hall, 1978.

[LDK+98] Stan Y. Liao, Srinivas Devadas, Kurt Keutzer, Steven W. K. Tjiang, and Albert Wang. Code optimization techniques in embedded DSP microprocessors. *Design Autom. for Emb. Sys.*, 3(1):59–73, 1998.

[LDK99] Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Design Autom. Electr. Syst.*, 4(1):12–38, 1999.

[LNV14] Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. In *Proc. 31st STACS*, pages 506–517, 2014.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[MN90] K. Mehlhorn and S. Nähler. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Inform. Process. Lett.*, 35(4):183–189, 1990.

[Moo56] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[NN13] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. In *Proc. 24th SODA*, pages 865–876, 2013.

[NS14] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Alg.*, 10(3):16, 2014.

[PD04] Mihai Pătraşcu and Erik D Demaine. Tight bounds for the partial-sums problem. In *Proc. 15th SODA*, pages 20–29, 2004.

[PT14] Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175, 2014.

[RRR01] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *Proc. 7th WADS*, pages 426–437. 2001.

[Ruž08] Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *International Colloquium on Automata, Languages, and Programming*, pages 84–95. Springer, 2008.

[SG06] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th SODA*, pages 1230–1239, 2006.

[Sim94] Imre Simon. String matching algorithms and automata. In *Results and Trends in Theoretical Computer Science*, pages 386–395. Springer, 1994.

[Slo] Neil James Alexander Sloane. On-line encyclopedia of integer sequences. https://oeis.org/A007814. Sequence A007814.

[SS78] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In *Proc. 10th STOC*, pages 30–39, 1978.

[SS82] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[Str00] Bjarne Stroustrup. *The C++ Programming Language: Special Edition (3rd Edition)*. Addison-Wesley, 2000. First edition from 1985.

[TISA16] Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. In *Combinatorial Algorithms: 27th International Workshop, IWOCA 2016, Helsinki, Finland, August 17-19, 2016, Proceedings*, volume 9843, page 213. Springer, 2016.

[Tro99] Zdeněk Tronìček. Operations on DASG. In *Proc. 4th WIA*, pages 82–91, 1999.

[Tro01a] Zdeněk Troníček. Episode matching. In *Proc. 12th. CPM*, pages 143–146, 2001.

[Tro01b] Zdeněk Troníček. *Searching subsequences*. Ph. D. Thesis, Department of Computer Science and Engineering, FEE CTU in Prague, 2001.

[Tro03] Zdenêk Troniĉek. Common subsequence automaton. In *Proc. 8th CIAA*, pages 270–275, 2003.

[TS05] Zdeněk Troníček and Ayumi Shinohara. The size of subsequence automaton. *Theoret. Comput. Sci.*, 341(1):379–384, 2005.

[vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6(3):80–82, 1977.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Switching and Automata Theory*, pages 1–11, 1973.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space θ(n). *Information Processing Letters*, 17(2):81 – 84, 1983.

[Wil00] Dan E Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.